The work was conducted under contract DASW01-94-C-0054, Task T-S5-1266, for the Defense Information Systems Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

PREFACE

This document was prepared by the Institute for Defense Analyses (IDA) under the task order, Object-Oriented Technology Implementation in the Department of Defense (DoD), in response to a task objective to develop strategies for the implementation of object-oriented technology (OOT) within specific information technology areas within the DoD. This document is one of a set of four reports on OOT implementation. The other reports, focusing on other areas of OOT, are IDA Paper P-3142, *Object-Oriented Development Process for Department of Defense Information Systems*; IDA P-3143, *Object-Oriented Programming Strategies for Ada*; and IDA Paper P-3145, *Software Reengineering Using Object-Oriented Technology.* All of this work was sponsored by the Defense Information Systems Agency.

The following IDA research staff members were reviewers of this document: Dr. Edward A. Feustel, Dr. Richard J. Ivanetich, Dr. Reginald N. Meeson, Dr. Judy Popelas, Mr. Clyde G. Roby, and Mr. Glen R. White.

Table of Contents

List of Figures

List of Tables

EXECUTIVE SUMMARY

Many of the current software engineering activities in the Department of Defense (DoD) center on migrating from obsolete legacy software systems to modernized migration systems. Legacy information systems incorporate obsolete technology such as closed systems, "stovepipe" design, and outmoded programming languages or database systems. Modernized migration systems are those systems, already in existence or being planned, that utilize or intend to utilize contemporary best practices in design and implementation. To date, transitioning a legacy system to a migration system has proven to be difficult.

Object-oriented technology (OOT) may be counted among the best practices for software development by virtue of its efficiencies in development and maintenance and its inherent support for reuse. OOT consists of a set of methodologies and tools for developing and maintaining software systems using software objects composed of encapsulated data and operations as the central paradigm. *Software wrapping* is a technique in which an interface is created around an existing piece of software, providing a new view of the software to external systems, objects, or users. Wrapping can be accomplished at multiple levels: around data, individual modules, subsystems, or entire systems.

This document describes the potential benefits, problems, and issues in using the OO technique of software wrapping in DoD information systems. It also describes the essential activities in OO wrapping, from determining the suitability of wrapping applications to implementing wrappers of legacy code or data using the Ada programming language.

Wrapping Basics

The narrow concept of a wrapped object is illustrated in Figure ES-1 on page ES-2, where the method icons surrounding the legacy software represent its encapsulation as a single object, accessible only through the object-defined methods (or operations). Any user access to the legacy software would be mediated through some of these methods, whether the user interface is a complex set of objects constituting a graphical user interface (GUI) or simple terminal line command input/output (I/O).

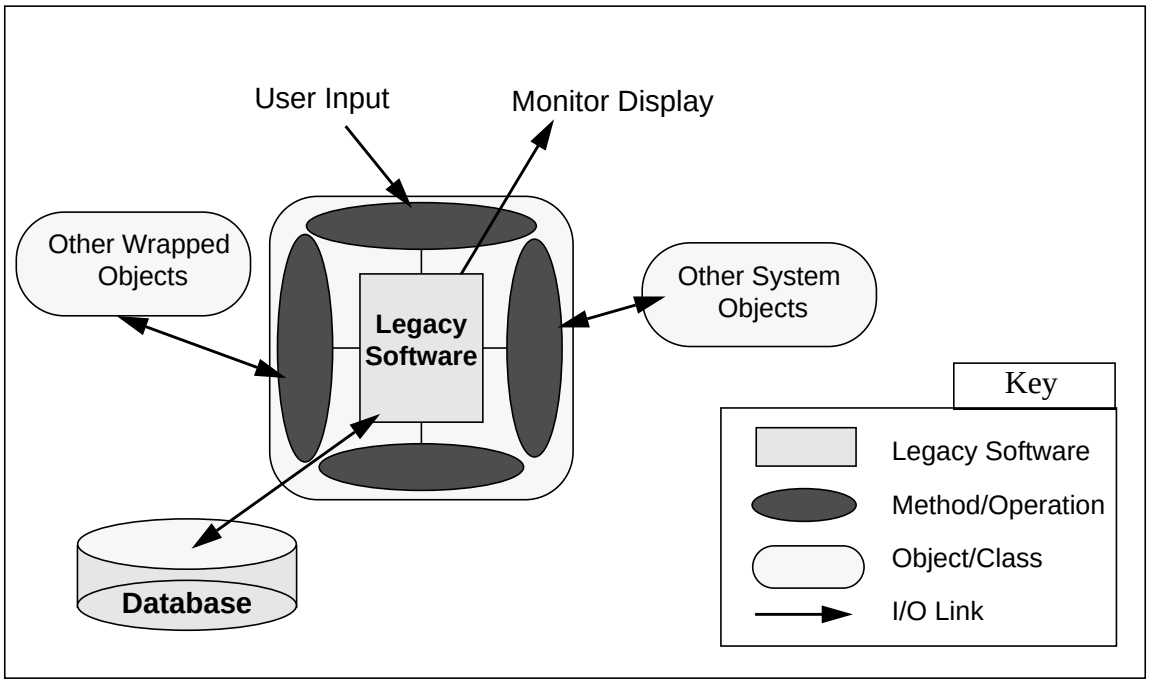

Figure ES-1. Legacy Software Wrapped as an Object

The broader conception of an OO wrapper is illustrated in Figure ES-2 on page ES-3, where an object model of multiple classes and objects is created as part of the wrapper to provide a natural OO interface to the principal conceptual entities implicit in the original system. The new objects and classes of such a wrapper can interface with the legacy programs and data in different ways. An application programming interface (API) may mediate communication between the wrapper object model and the legacy program, as illustrated in Figure ES-2. When the legacy software is a database, a database server might provide the functionality of an API, with objects accessing the database through SQL calls to the server.

System Migration Strategies

Wrapping fits into the following broader strategies for entire system migrations: divide-and-conquer, divide-and-wrap, unite-and-conquer, and one-shot rebuild.

Divide-and-conquer. This strategy proceeds incrementally, dividing legacy subsystems and applications into those selected for immediate conversion to OO technology and those that are not. The most suitable candidates are converted and integrated with the existing systems, and the process is repeated until either the entire system or all suitable parts of it are converted to OO form. This supports a staged transition, ordinarily more manageable, and involves lower risk than attempting to convert an entire system at once. Wrapping could participate in this

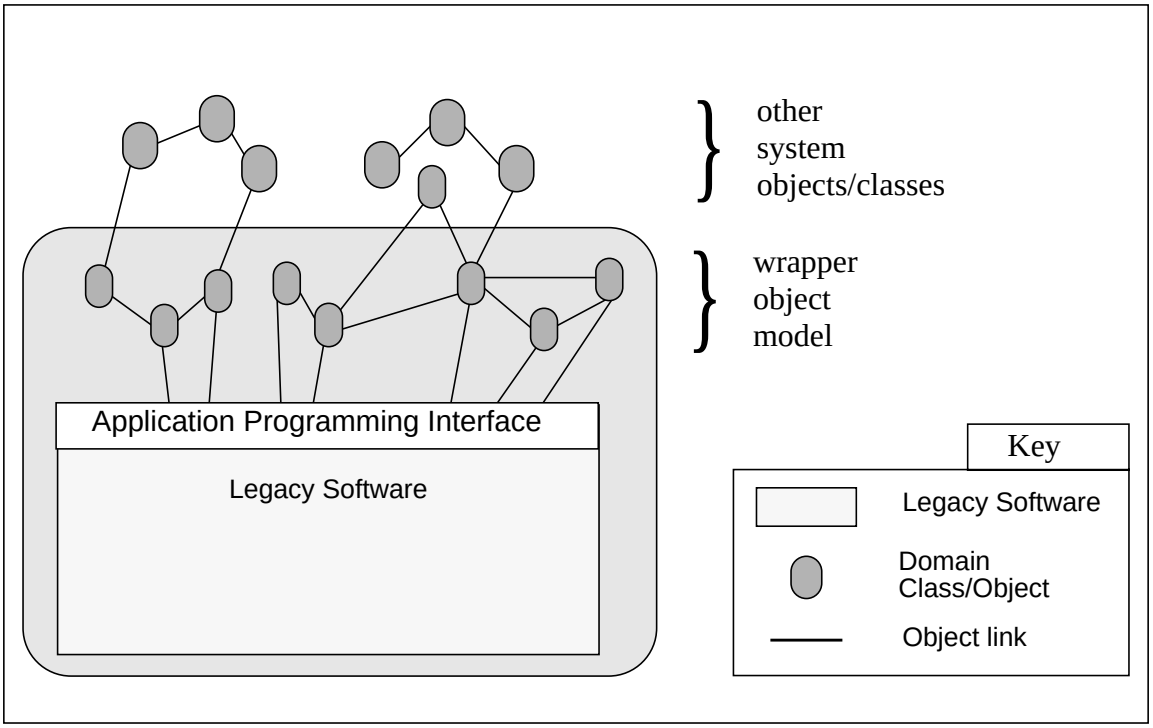

Figure ES-2. Legacy Software Wrapped with Object Model

strategy by providing temporary modernization of some system components to ease their integration with fully converted components. Leaving some system components untouched, however, can leave some integration difficulties, and a mixture of traditional and OO system components could be awkward to support throughout the duration of incremental modernization. This is a potential drawback to any divide-and-conquer scheme, whether or not wrapping is involved. However, wrapping can help alleviate this problem, as demonstrated by the next general strategy, which we call "divide-and-wrap."

Divide-and-wrap. Wrapping can be used quite broadly to effect a complete conversion of the legacy system to OOT in a single step by wrapping everything that is not fully reengineered. Wrapped components could then be incrementally reengineered, as feasible, using OO techniques. This strategy eases integration of all the pieces at different stages of transition since the methods interfacing wrapped objects can perform any necessary translations between legacy and modernized components. It offers flexibility in scheduling the transition increments through variations in both the amount of reengineering and the granularity of the components wrapped. In some cases, whole subsystems may be wrapped for a lower-cost transition stage, while wrapping may be executed at a finer level during transition stages when more time and staff resources are available. The principal drawback to this general transition strategy is that

wrapping large portions of a system may require considerable rework (of object hierarchies, methods, and data structures) when these components are unwrapped and decomposed into more meaningful objects.

Unite-and-conquer. This strategy achieves a unification of system applications and databases through a common OO framework that organizes access to legacy code and data as well as to new and reengineered OO system components. Such a framework can be constructed as part of developing business or enterprise models of the business activities supported by the legacy information system. However, developing business models can be a time-consuming analysis task for large systems since the essential business objects must be identified and mapped to the relevant existing programs and/or databases. Thus, a unite-and-conquer strategy can only be effectively executed at a migration stage when sufficient resources are available for this extensive analysis. When the resources are available, the payoff can be considerable in later stages of migration.

One-shot rebuild. Multiple experiences in building large OO systems indicate that, for OO systems in particular, incremental development is more effective than the classic waterfall development model. While a one-shot waterfall development has never been recommended for OO systems, it can be feasible in smaller automated information systems to apply locally incremental development to the system as a whole. One-shot rebuild could also be viable for a large legacy systems if it is very similar to an existing OO system that has already been implemented, or if it can be constructed out of existing tested frameworks and repository objects.

Issues and Findings

What are the different methods of software wrapping and which are preferable? When the resources are available, domain object models composed of multiple related domain objects are preferable for wrapping legacy components rather than simply wrapping each component as an isolated object. Such object model wrapping provides a better foundation for any subsequent legacy modernization or extensions. The costs of object model wrapping can be minimized by judicious abstraction of the domain object classes, modeling only those features essential to wrapping.

What criteria should be used in selecting legacy software components for wrapping? Election of a wrapping strategy and selection of components for wrapping require good reasons to wrap rather than to reengineer and a determination of feasibility of wrapping. These reasons include the need for rapid modernization in the absence of sufficient time or staffing resources for reengineering. Another set of reasons consists of various barriers to effective reengineering,

such as the absence of documentation or available domain experts, and the complexity or great volume of legacy code. The feasibility of wrapping depends on various features of the legacy system and target system environments, being improved by modularity in legacy code, and ready support for interfaces between legacy components and the target OO environment. Under such a favorable environment, software wrapping can provide the most effective means of meeting modernization deadlines.

What overall system migration strategies are least risky and how might they incorporate wrapping? The "one-shot rebuild" strategy is widely considered risky for large systems because it attempts too much reengineering in a single step. In contrast, the "unite-and-conquer" strategy is considered superior and is recommended because it uses a unifying object model of the system domain to wrap legacy components, supporting a natural incremental modernization. This strategy minimizes costly revisions to the object models by developing the basic domain object model at the outset and unifying the modernized and legacy components with it. The advantages of "unite-and-conquer" strategy do come at the expense of additional upfront costs in building the domain (or business) object model, compared to the "divide-and-conquer" approach which only develops those parts of the domain object model that are needed for the modernized parts of the system at any particular stage of migration. Thus, "unite-and conquer" is only recommended for a migration stage when there are sufficient resources available for a full domain analysis and object model development.

What programming techniques are involved in implementing wrapping? Two alternative techniques for implementing the interface between an object wrapper and legacy software are described for implementations in the first Ada programming language standard (Ada 83): direct calls to legacy procedures and functions using the interface pragma to a legacy language, and indirect calls via an operating system or via an intermediate language. While direct calls are preferable for accessing legacy procedures, this is not always possible due to environment-specific barriers. The later standard, Ada 95, has added several new features to greatly facilitate the interface to code in foreign languages, which should avoid any need for indirect methods in many cases.

When wrapping an SQL database, there are three viable options for implementing an Ada binding to SQL: all-Ada binding, embedded SQL, and use of an additional programming language, such as SAMeDL (SQL Ada Module Description Language).

In addition, a number of unresolved wrapping issues were identified:

- What general guidelines are appropriate for the transition from a mainframe-based legacy system to a local area network based client-server model?
- What guidelines can be provided for mapping the legacy terminal I/O into today's GUIs?
- What standards-based support can be provided for interfacing OO programs in Ada (and other languages) to database management systems, whether relational or object oriented?
- What guidelines can be established for selecting techniques for promoting interoperability among different DoD information systems? For example, is the Object Management Group's Common Object Request Broker Architecture suitable for this purpose?
- What are the unique issues related to wrapping and migration of real-time systems?

1. INTRODUCTION

1.1 PURPOSE AND SCOPE

The purpose of this document is to support the migration of legacy Department of Defense (DoD) information systems by providing a detailed explanation of the potential benefits of using the object-oriented (OO) technique of software wrapping as a mechanism. It specifically describes the risks, problems, and issues in the use of OO wrapping techniques for DoD information systems. It is also intended to be a guide to all the essential activities in OO wrapping, from determining the suitability of wrapping applications to implementing wrappers of legacy code or data using the Ada programming language. This report is intended to address issues of interest to anyone interested in techniques for facilitating the software migration process.

1.2 BACKGROUND

Much of DoD's current software engineering activities center around the migration from obsolete legacy software systems to modernized migration systems. *Legacy information systems* are those systems currently operating that incorporate obsolete technology such as closed systems, "stovepipe" design, and outmoded programming language or database systems. *Modernized migration systems* are those systems already in existence or are being planned that utilize or intend to utilize contemporary best practices in design and implementation. OO technology (OOT) may be counted among the best practices for software development by virtue of its efficiencies in development and maintenance and its inherent support for reuse, as explained in a companion report [IDA95a]. Transitioning a legacy system to a migration system has proven to be difficult; in response, several effective strategies are described in this report that would facilitate migrating to a modernized system.

1.3 ORGANIZATION OF DOCUMENT

Chapter 2 reviews the concepts of wrapping software components and includes an example from an actual software migration project to illustrate these wrapping concepts.

Criteria are identified for evaluating the suitability of a wrapping strategy and for selecting components for wrapping.

Chapter 3 places wrapping in the broader context of alternative migration strategies for a whole system, arguing the advantages of the “unite-and-conquer strategy” using a unified object model throughout progressive stages of migration, as compared to the other three strategies (divide-and-conquer, divide-and-wrap, and one-shot rebuild).

Chapter 4 discusses wrapper types and content, and the wrapping of software components at different levels of granularity.

Chapter 5 describes alternative OO wrapping techniques that provide encapsulation of legacy code or data during migration to a modernized system.

Chapter 6 discusses wrapping implementation. Several examples of wrapping using Ada interface pragmas are given for functions or subprograms written in the Cobol, C, and Fortran languages. A simplified scenario is then presented of a legacy migration situation as a basis for illustrating wrapping techniques. Details are provided on wrapping a data file and a program from the legacy system using Ada interface pragmas. Complete code for this example is provided for reference in Appendix A. The general issues of OO programming in Ada are not analyzed in this document, but do receive detailed treatment in a companion report [IDA95b].

Chapter 7 summarizes the basic guidelines for the application of wrapping and identifies the remaining issues involved in the implementation of wrapping.

References, glossary, and acronyms are provided at the end of the document.

2. WRAPPING CONCEPTS

Software wrapping is a technique in which an interface is created around an existing piece of software, providing a new view of the software to external systems, objects, or users. Wrapping can be accomplished at multiple levels: around data, individual modules, subsystems, or entire systems. This chapter provides a general introduction to the types of software wrapping with additional details and examples in the subsequent chapters.

2.1 WRAPPING SOFTWARE COMPONENTS AS INDIVIDUAL OBJECTS

The narrow concept of a wrapped object is illustrated in Figure 1 on page 4. The method icons surrounding the legacy software represent its encapsulation as a single object, accessible only through the object-defined methods (or operations). Any user access to the legacy software would be mediated through some of these methods, whether the user interface is a complex set of objects constituting a graphical user interface (GUI) or simple terminal line command input/output (I/O). Other system objects, and even other wrapped objects, also access the wrapped legacy software only through the wrapper's methods. Access from the wrapped object to other parts of the system may still occur directly from the legacy code to databases and user interfaces, or may be mediated by calls to the methods of other system objects. We refer to such wrapping of legacy software into a single object as *single-object wrapping* or *direct wrapping*. Direct wrapping of different types and at different levels of granularity can be created through different partitions of a legacy software system's functions, programs, and databases, as described in detail in Chapter 4.

2.2 WRAPPING SOFTWARE WITH OBJECT MODELS

The broader concept of an OO wrapper is illustrated in Figure 2 on page 4. An object model of multiple classes and objects is created as part of the wrapper to provide a natural, OO interface to the principal conceptual entities implicit in the original system. The new objects and classes of such a wrapper can interface with the legacy programs and data in different ways. An application programming interface (API) may mediate communication between the wrapper object model and the legacy program, as illustrated in Figure 2

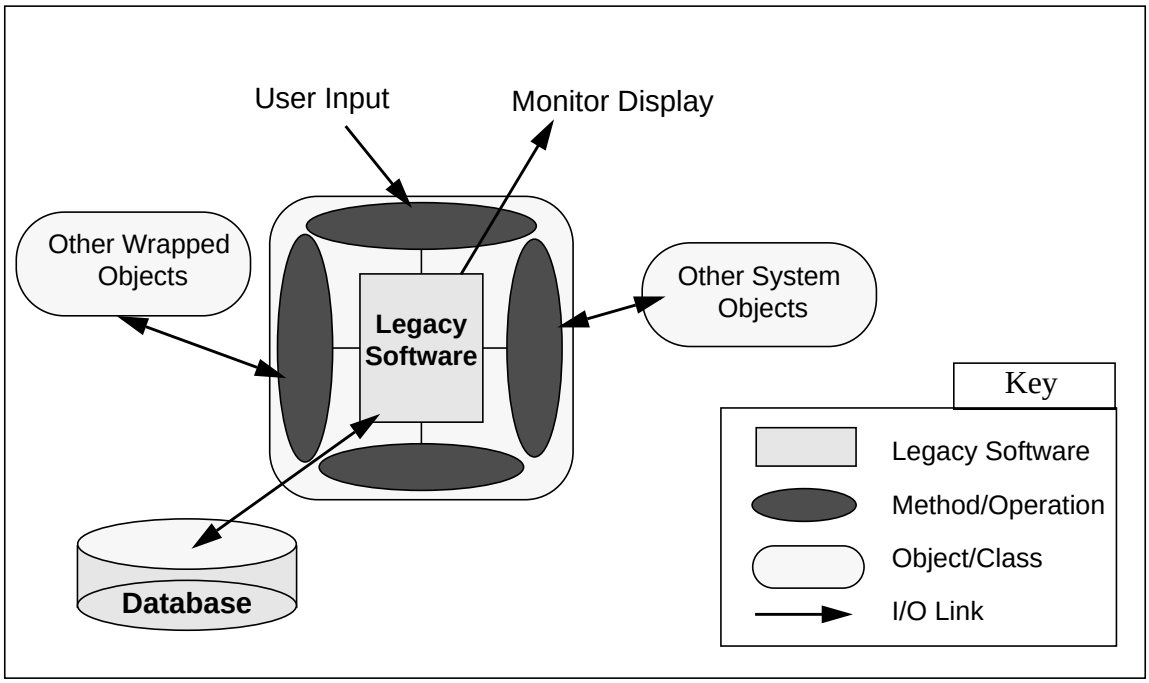

Figure 1. Legacy Software Wrapped as an Object

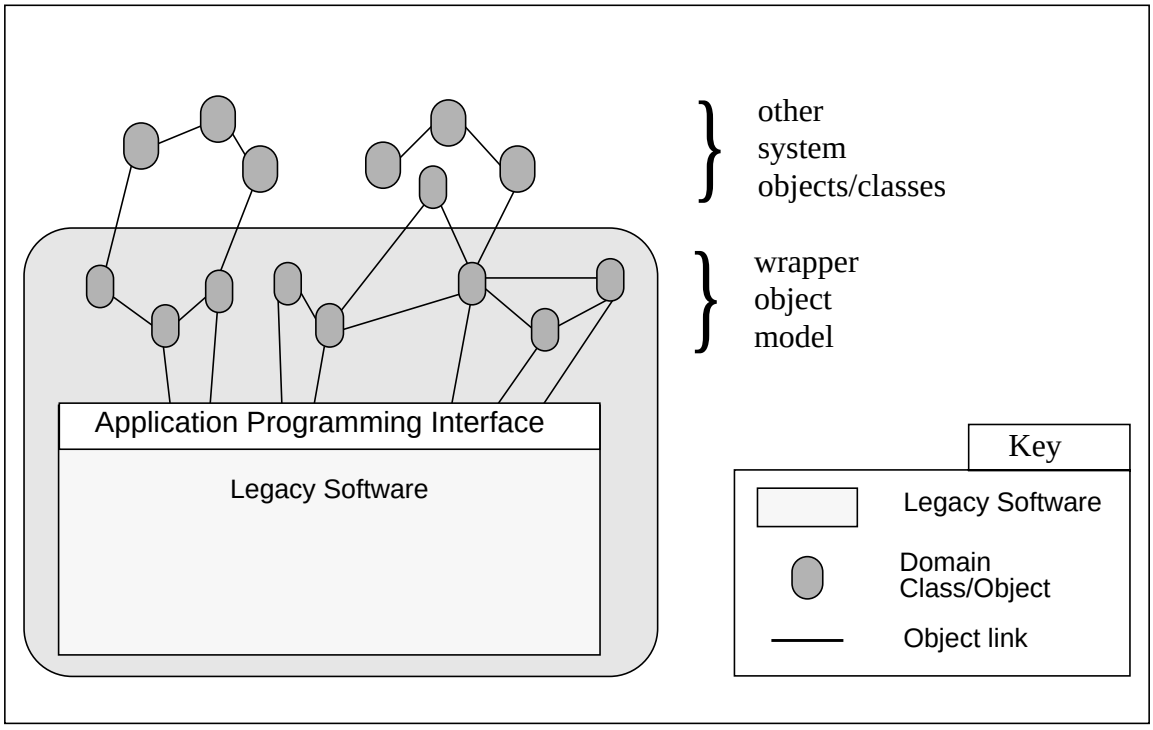

Figure 2. Legacy Software Wrapped with Object Model

on page 4. When the legacy software is a database, a database server might provide the functionality of an API, with objects accessing the database through SQL calls to the server. When a separate API is used, it might be written in the OO programming style of the modernized portion of the OO system, e.g., the API, itself might be an object. Making an API into an object effectively wraps the legacy software as a single object. However, when this wrapper is combined with an object model of the legacy software, a much richer interface is created than what appears in a simple "direct wrapper." Alternatively, the API might be composed simply of minor modifications to legacy code to support external access directly from wrapper objects.

However the legacy software is interfaced, the essence of the object model wrapping approach is the interface through multiple classes/objects which are natural parts of an object model of the application. This provides significant advantages over simply directly wrapping software as an individual object: application objects will persist throughout subsequent migration stages while wrapped software objects will need to be replaced when their legacy software is modernized. These advantages are expanded upon in the discussion on wrapping costs and benefits (Section 2.4 on page 7).

One of the earliest examples of object model wrapping is found in the geometric modeling application described in [DIET89]. In the migration described, an API was composed by modifying selected subroutines in the legacy system so that the object classes of the modernized components of the migration system could access them. Object classes of geometric models were defined in an OO programming language, connected to corresponding routines via the API, and instantiated to instances at run-time. This example is described in some detail in the next section.

2.3 EXAMPLE OF OBJECT MODEL WRAPPING

An alternative OO interface to legacy system wrapping was pioneered at IBM's Thomas J. Watson Research Center in the Tiered Geometric Modeling System (TGMS) [DIET89]. This system provided an alternative OO interface to a legacy system, the Geometric Design Processor (GDP), a solid modeling system composed of several hundred thousand lines of PL/I code [WESL80, WOLF87]. TGMS used a set of objects in the AML/X object-oriented programming language [NACK86] to wrap the entire GDP system. The relationships between these systems are illustrated in part by Figure 3 on page 6.

This figure provides an informal representation of part of the object model in TGMS that is used in wrapping the functionality of the GDP system. The hierarchy of geometric

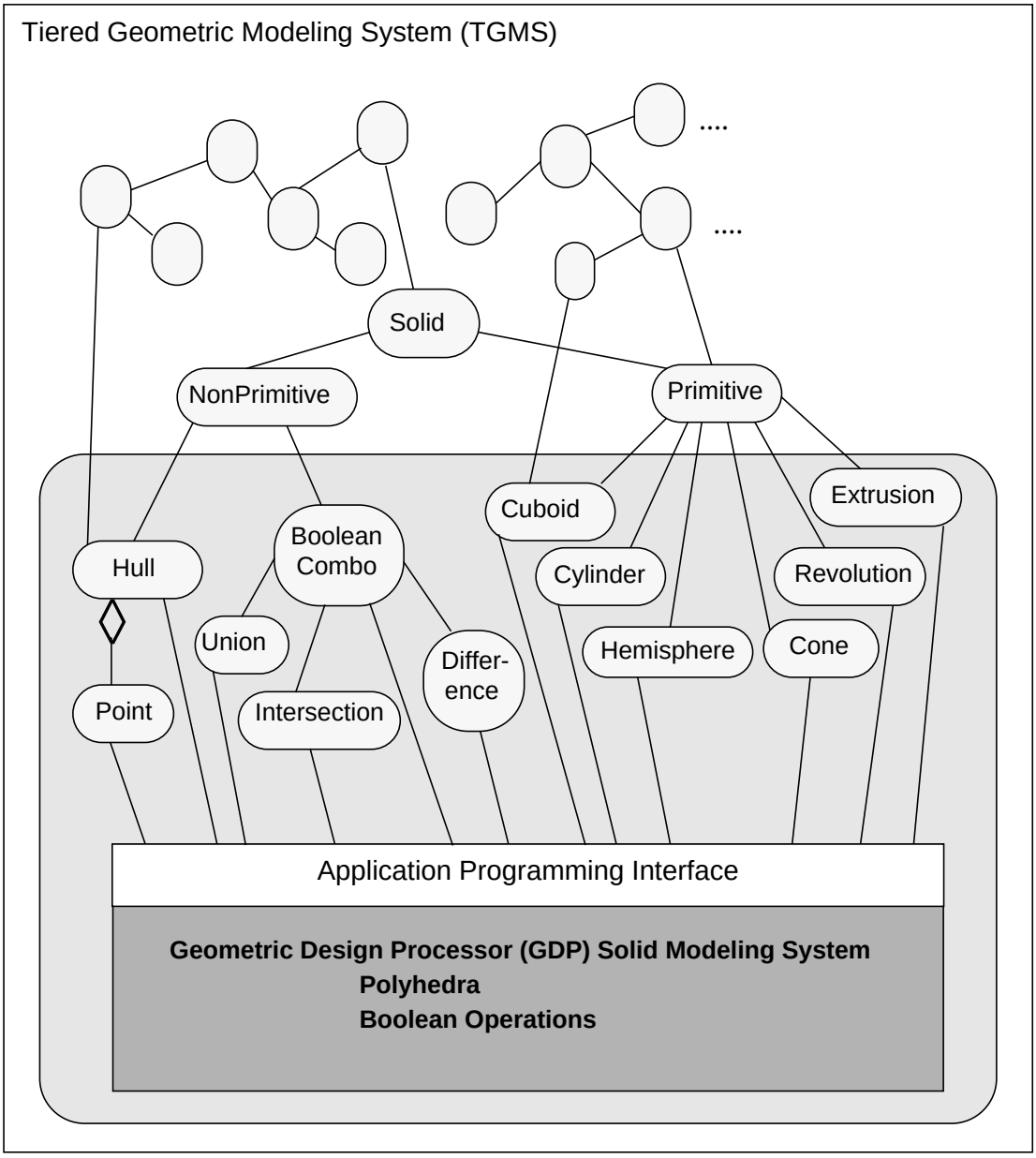

Figure 3. Example of System Wrapping for a Geometric Modeling System

models is shown by straight line links between class icons, starting with the *Solid* class. The primitive types of solid objects are shown to be cuboids, cylinders, hemispheres, cones, revolutions, and extrusions. The non-primitive types are hulls, composed of the convex hull of a set of points, and boolean combinations of other solids. Only those objects that interact directly with the wrapped software (or its API) are shown as included within the scope of the wrapper (in the shaded rounded box). Connections between wrapper objects and the

legacy code API are shown as simple straight line segments connecting object icons to the API box. Higher-level objects, such as the generic *Solid, Primitive*, and *NonPrimitive,* are shown as part of TGMS but not as part of the wrapper under the assumption that they do not have such direct connections. Although the actual cutoff between wrapper objects and others in TGMS was not clear from our source, the idea of this distinction is well illustrated since it is quite possible that only the more specific object classes would be connected to the legacy code implementing their functionality. Other non-wrapper objects of TGMS are alluded to by the unlabeled object bubbles. These objects may include additional functionality and interface and control objects.

2.4 ADVANTAGES AND DRAWBACKS OF WRAPPING

2.4.1 General Advantages of Wrapping

Once wrapped, legacy software can function as a set of objects or classes within a larger OO system, interfacing via message passing.

> *Wrapping can establish compatibility of old code with new interoperable data description standards by supporting the translation between these standards and legacy formats at the methods interface to the wrapped software.*

The concept of using a wrapper to establish data description compatibility is illustrated in Figure 4 on page 8, where the legacy software is depicted as having the format F1, the standardized format of the modernized OO portion of the migration system is F2, and the API performs the mapping back and forth between them.

> *Wrapping facilitates rapid transition of multiple legacy systems to fewer and more interoperable systems, thereby improving interoperability and reducing maintenance and modification costs.*

Wrapping facilitates the rapid transition from legacy to migration systems by minimizing the amount of code rewriting and database restructuring required in the initial stages of migration. Thus, a partially modernized migration system may be fielded sooner than it could if the entire legacy system were reengineered at once. Maintenance and modification costs can be more quickly reduced since there are fewer separate systems to maintain when multiple redundant legacy systems are transitioned to fewer standardized systems.

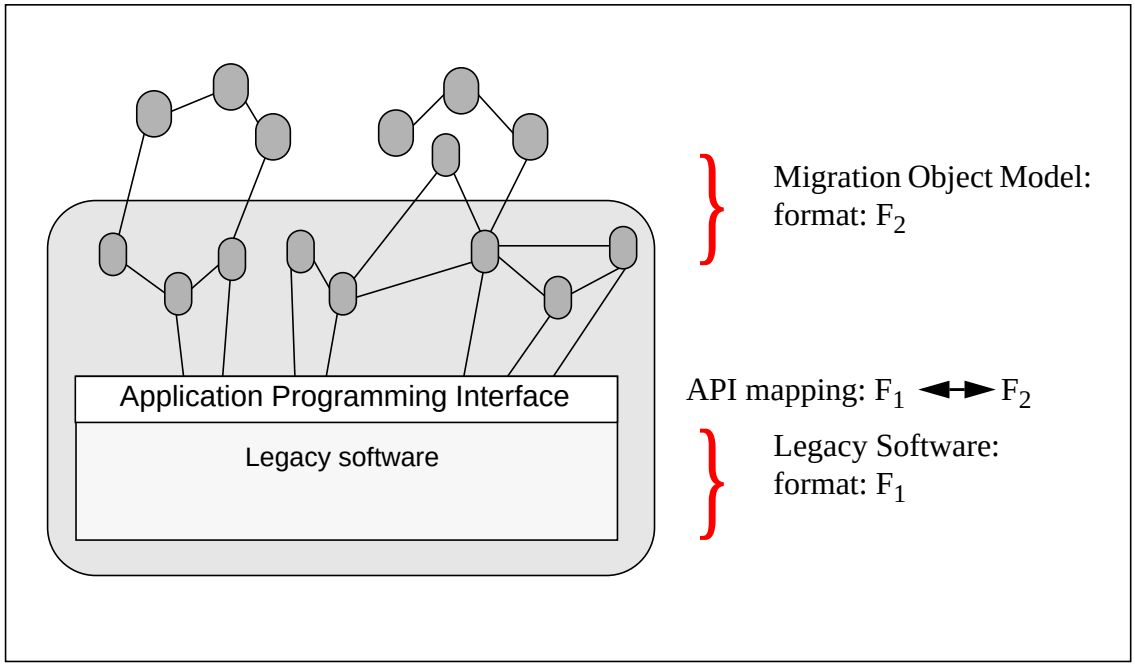

Figure 4. Wrapping Supports Translation to Standard Data Element Formats

However, migration systems containing wrapped components still face some of the maintenance headaches of the legacy systems since the legacy code is likely to be difficult to modify. Object wrappers can ease some of the modification burdens of software maintenance in so far as the modifications can be accomplished through specialization of a wrapper class without modifying the original code. But the feasibility of this technique will vary depending on the system. However effective this technique might prove, bugs in the legacy software will ordinarily require direct modifications, assuming the code is not reengineered. Thus, maintenance requiring modification of legacy code cannot be entirely avoided in systems using wrapping.

Wrapping is most often discussed as a temporary measure—ordinarily as an interim solution to the problems of modernizing legacy software, a stepping stone on the path of a full system reengineering. In some cases it may also serve as a terminal treatment for obsolescent software, when interoperability is required temporarily for a system that is close to retirement. In either case, it can provide some of the benefits of OO systems without the costs of fully reengineering all the legacy code. The wrapped sections of a legacy system can participate as objects in a broader OO system, while the details of the legacy code and/or data are encapsulated. When data are so encapsulated, the system may be accessed via standard modernized data and object definitions without disrupting the legacy database.

Wrapped objects may even be reused in other systems if the granularity wrapping creates objects of potential use elsewhere.

2.4.2 Object Model Wrapper Advantages

Wrapping with object models deserves special recognition as a possible means of encapsulation distinct from direct wrapping of a legacy components as a single object. Fortunately, such object models need not contain the full detail expected of a reengineered system in order to provide structured encapsulation of legacy databases, code, or both. An initial model of an enterprise, business, or other application domain structure need not include much, if any, of the functioning methods that would implement business or application operations. When functioning as a database interface, such a model may consist primarily of objects with pointers to (or access scripts for) their corresponding instances and attributes in the legacy data base. Operational aspects of these models can continue to be performed by legacy programs until they are transitioned to the relevant objects or to more specific instances of them in reengineered code. Thus, such object models may be developed and incorporated in a migration system with much less effort than required by a complete reengineering of the covered automated information system (AIS) activities.

Wrapper object models provide an extraordinarily useful approach to encapsulation due to the rich object structure within which they accomplish it. Object models can encapsulate a database at much finer granularity than a single object wrapper can achieve. The separate elements (objects) of an object model can participate in a structure of inheritance and service relations that illuminates the essence of the encapsulated component, as contrasted with the mere data hiding found in the unstructured encapsulation of direct wrapping. Wrapper object models provide a structural model for the objects implicitly handled by legacy code and data. Unlike a simple single object wrapper, object models are not "throw-away" code, discarded after the encapsulated components are modernized. Object models can provide a lasting foundation of domain objects which can be reused in subsequent migration stages. Rich elaborations of the initial class attributes and operations can be constructed in future migration steps to flesh out the details hidden in the legacy code as it is modernized. In such transitions, any hierarchies contained in a wrapper object model can be enriched through inclusion of more specific subclasses whose operations incorporate functionality previously accommodated by the wrapped legacy code. Such objects may, of course, be reused, both in other migration systems, and as foundations for rapid development of new applications.

In short, the use of object models for encapsulation of legacy system components has many of the advantages of OOT in general, while its costs can be minimized in its initial application through judicious abstraction of only the essential features of the model objects.

The example outlined in Section 2.3 on page 5 for object model wrapping in TGMS exhibits these advantages. TGMS can be easily extended with new classes of solid models through their addition to the TGMS OO hierarchy. Objects in these classes may then be combined with existing solid models using the existing wrapped boolean combination operations from the GDP legacy system. The wrapped capabilities of TGMS may also be easily extended by specialization of the legacy object functionality using OO inheritance. New classes might be added as specializations of revolution and extrusion classes, for example. As with any OO subclass, such specializations can utilize any applicable operations in their existing superclasses. Thus, the wrapped legacy code of GDP is readily extended with new functionality. If the GDP code had been wrapped more directly as a system or as individual functions and data, such natural extensions would not be so straightforward to implement.

2.4.3 Direct Wrapping Drawbacks

A drawback specific to the direct wrapping technique is the scale of large legacy components when wrapped as objects. If a legacy software component of a large size were to be fully reengineered using OOT, it would most likely break down into multiple interacting objects to better reflect its implicit organization of information and procedures. Direct wrapping can save some of the work of this decomposition and reorganization in the short term when wrapping at a coarse level using large software components. But such savings come at the cost of a coarser scale representation that may be more awkward to integrate with the rest of an OO system. Where other objects might only need to access some small part of the data or functionality of the wrapped object, they must refer to the wrapped component as a whole. Where other systems may only need minor functions from a wrapped component, they will have to incorporate the whole component if they are to benefit from its reuse. Furthermore, subsequent reengineering of a directly wrapped legacy program may require substantial rework of an existing class hierarchy in order to accommodate the new classes abstracted from a legacy program.

Wrapping legacy software components directly as objects may share some of the advantages of wrapping with object models if the direct wrapping is performed at a fine enough granularity, such as the level of individual functions. Such fine granularity allows the reuse of function objects without the unneeded baggage of the rest of a system. But wrapping legacy functions as objects remains unlikely to produce a natural and lasting object model of the system domain. Functions ordinarily map more naturally into the methods of an OO system than into its objects. Effective OO design typically requires taking a fresh look at the system in order to abstract the relevant objects from its requirements and functionality. This fresh perspective can be difficult to achieve if restricted to creating objects from direct wrappings of legacy components. While direct wrapping of legacy system components can often be accomplished more quickly than wrapping with multiple newly abstracted objects, the latter approach can provide a more stable class/object structure throughout subsequent migration phases.

2.4.4 General Drawbacks of Wrapping

The principal drawback to any type of wrapping is in long-term maintenance, since the legacy software remains beneath the wrappers, and is likely to be difficult to maintain or modify.

This drawback can be substantially mitigated if future modifications can be implemented within the OO portion of a migrated system, external to the legacy code. The addition of new methods and operations to a class wrapper, for example, may be accomplished without touching the legacy code. Augmentation of the methods, attributes, or both of a class wrapper might also be achieved independently of the legacy code by creating new subclasses with the additional structure and/or functionality. Maintenance involving fixes to bugs in the legacy code may be more difficult, however, requiring direct modifications of the legacy code itself. These potential problems with maintenance and modification are reasons for considering wrapping as only a temporary solution for modernizing software systems, with full reengineering (or obsolescence) as an eventual goal (or expectation).

For all these reasons, it is preferable, when the resources are available, to wrap legacy software with a carefully abstracted object model than to directly wrap a whole program or system as a single object.

2.5 WRAPPING CRITERIA

The principal criteria for selection of some component of a legacy system for wrapping are whether it is feasible to wrap and whether there are good reasons to wrap as opposed to reengineering or replacing it.

Feasibility of wrapping depends primarily on how modular the legacy system component is and how readily it can be accommodated within the migration target environment. If a segment of code and/or data are fairly self-contained, with relatively few types of calls out to and in from external code, then it may be feasible to wrap it efficiently. When a segment of application code is interwoven with complex input/output (I/O) to users, data stores, other code, and other applications, it may be as costly to wrap as to reengineer, so that reengineering is preferable for its more thorough modernization. Even a very modular system component can be difficult to wrap if its elements are not well supported within the targeted migration environment. Code written in a proprietary language for obsolescent hardware or operating systems, for example, may have no support on a modern computing platform. Obsolete database management systems may also prove impractical to port to new platforms. Thus, wrapping feasibility must take into account both the modularity of the legacy component and its support within the target migration environment.

A major reason for considering wrapping some parts of legacy systems is a situation in which it is especially difficult to fully reengineer that part of the system, but strong pressures exist for modernizing the whole system quickly. Some components of a legacy system can be especially difficult to fully reengineer due to a variety of factors, such as the following:

- Absence of documentation
- Departure of all domain experts
- Complexity of code
- Fragility (or brittleness) of code
- Size of code or database
- Staffing resource limitations

At the same time there may be strong pressures to modernize the existing legacy system to meet pressing deadlines due to factors such as the following:

- Expiring hardware and software contracts
- Shift to new platforms
- Requirements for interoperability with other reengineered AISs
- Data item standardization requirements

Under conditions like these, wrapping may be the most effective short-term means of meeting interim modernization deadlines.[1]

Consider, for example, a hypothetical legacy information system written in Cobol, hosted on IBM mainframes whose maintenance contract is up for renewal in 10 months. Imagine that an analysis has shown that the total costs for hardware to replace the aging IBM mainframes with workstations under a client-server architecture are much less than the current yearly maintenance costs. Such discrepancies between the costs of replacement versus maintenance of obsolete system hardware are not uncommon in such transitions. Suppose, further, that this system is also required to be interoperable with several other information systems that have been recently transitioned to OOT and are interacting using the CORBA (Common Object Request Broker Architecture) distributed OO protocols. In addition, budgetary constraints project decreasing funds for maintaining the same basic functionality of this system augmented with the additional interoperability requirements. Thus, there are budgetary pressures for a transition to a system of lower operating costs, there are time pressure costs to effect the transition before the old hardware maintenance contract must be renewed, and there is some reason to consider a transition to OO technology in order to facilitate meeting interoperability requirements. In such a context, OO wrapping techniques may be the key to transitioning the legacy system to a partially modernized one within the given constraints on budget and within the time constraints.

Some system wrapping may be the most effective strategy in cases in which, although the reengineering is not especially difficult, the time pressures and resource limitations do not allow full reengineering within a required interim modernization timeframe. Alternatively, some whole systems may be so large or complex that reengineering may be too daunting a task to tackle all at once. In such situations, wrapping of subsystems could provide the basis for an incremental reengineering in which decomposable subsystems and components are first identified and wrapped with standardized interfaces, followed by incremental reengineering of these wrapped objects.

[1] Other techniques, such as code translation, or database modernization, may also be effective for interim modernization of some legacy system components, as discussed in [BLSM93].

Another situation in which wrapping may be indicated arises when a legacy system, or some portion thereof, is expected to become obsolete within the relative near term. In such cases, a full reengineering effort may be a waste of resources, since the system might no longer be needed by the time it could be reengineered. But it may not be possible to leave the legacy system completely unaltered for its remaining lifetime due to incompatibility with interacting AISs that are modernized prior to its termination. Thus, wrapping could be a cost-effective means of ensuring short-term compatibility without wasting the efforts of reengineering obsolescent software.

3. SYSTEM MIGRATION STRATEGIES

In this chapter, we view the migration process as a whole and examine how wrapping may fit into four broader strategies for entire system migrations, before getting into the details of different kinds of wrapping. The four strategies are divide-and-conquer, divide-and-wrap, unite-and-conquer, and one-shot rebuild. In the next chapter, we focus more locally on different types of components in legacy systems, describing how wrapping may be used to facilitate their migration to a modernized system.

3.1 DIVIDE-AND-CONQUER

One general strategy for AIS migration, called "divide-and-conquer" [TAYL92], proceeds incrementally, dividing legacy subsystems and applications into those selected for immediate conversion to OOT and those that are not. The most suitable candidate or candidates are converted and integrated with the existing systems, and the process is repeated until either the entire system or all suitable parts of it are converted to OO form. This supports a staged transition, which is ordinarily more manageable and involves lower risk than attempting to convert an entire system at once.

Wrapping could participate in this strategy by providing temporary modernization of some system components to ease their integration with fully converted components. Figure 5 on page 16, for example, illustrates a divide-and-conquer modernization increment for a legacy system consisting of four application programs, AP1, AP2, AP3, and AP4, and three databases, DB1, DB2, and DB3. Application program AP1 is wrapped whole as a single object. AP2 is completely reengineered as an OO program. The databases, DB1 and DB2, are both wrapped whole to provide suitable interfaces for data access from the reengineered AP2. AP4 is slightly modified to access its data through the wrapper of DB2. AP3 and DB3 are left unchanged.

However, leaving some system components untouched, such as AP3 and DB3, can pose some integration difficulties, and a mixed traditional and OO system components could be awkward to support throughout the duration of incremental modernization. This is a potential drawback to any divide-and-conquer scheme, whether or not wrapping is

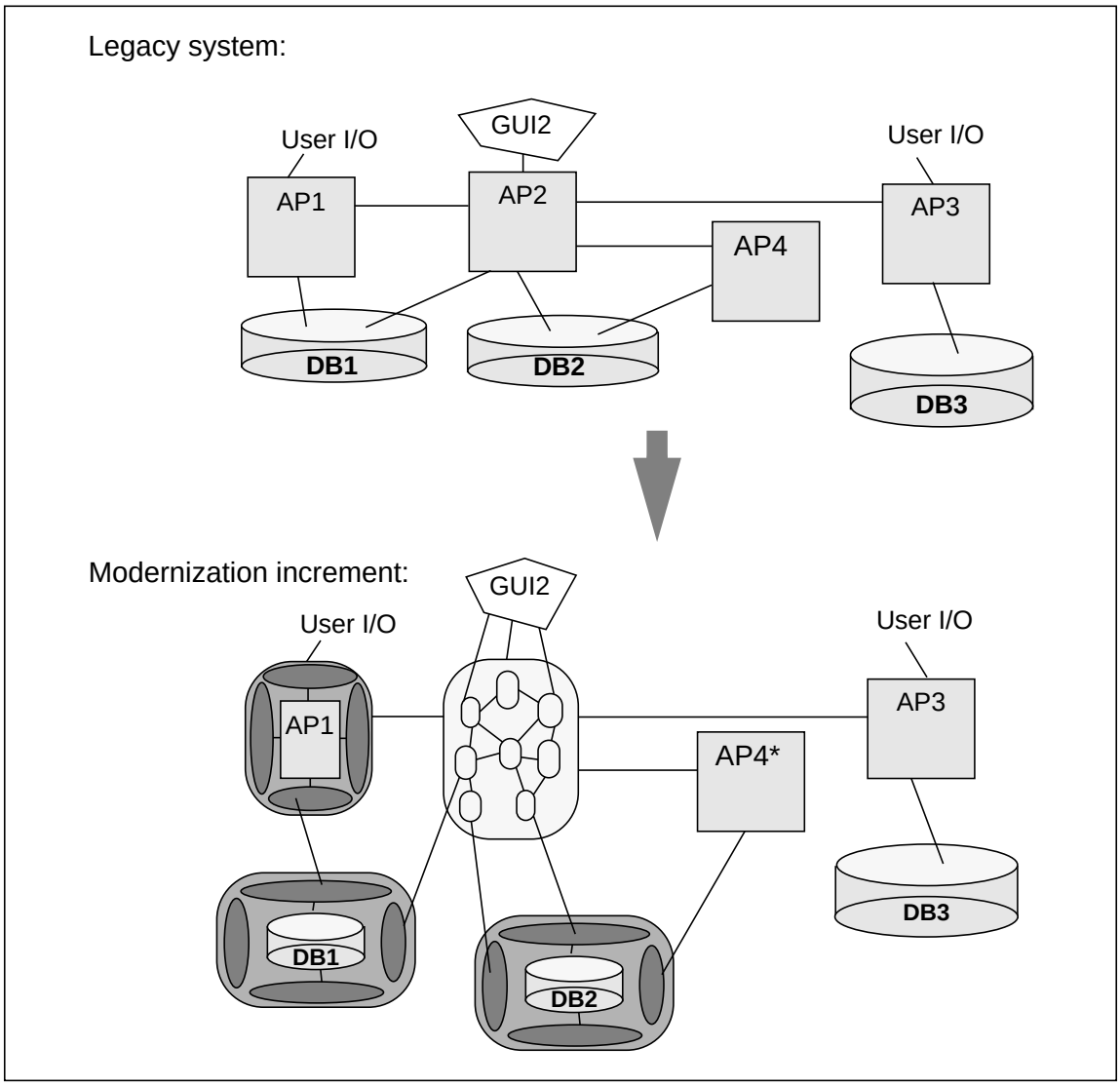

Figure 5. Divide-and-Conquer With Some Wrapping

involved. However, wrapping can help alleviate this problem, as demonstrated by the next general strategy, which we call “divide-and-wrap.”

3.2 DIVIDE-AND-WRAP

Wrapping can be used quite broadly to effect a complete conversion of the legacy system to OOT in a single step by wrapping everything that is not fully reengineered. Wrapped components could then be incrementally reengineered, as feasible, using OO techniques. Figure 6 on page 17 illustrates this strategy by wrapping all applications and data except a single application, AP2, which is fully reengineered. In this example, one application program, AP1, is wrapped and supplemented with a GUI and an additional

object/class structure, as shown. Two databases, DB1 and DB2, are wrapped separately. One subsystem, consisting of database DB3, application program AP3, and its user interface, is wrapped in its entirety, augmented with a GUI and reconnected through its new wrapping methods to the reengineered application AP2 and to external information systems. One application, AP4, is simply wrapped and reconnected via methods to its connected database DB2 and a calling application (the reengineered AP2). This migration system then consists of objects of a wide range of granularity, from simple GUI objects and domain objects to program objects, database objects, and a whole subsystem object.

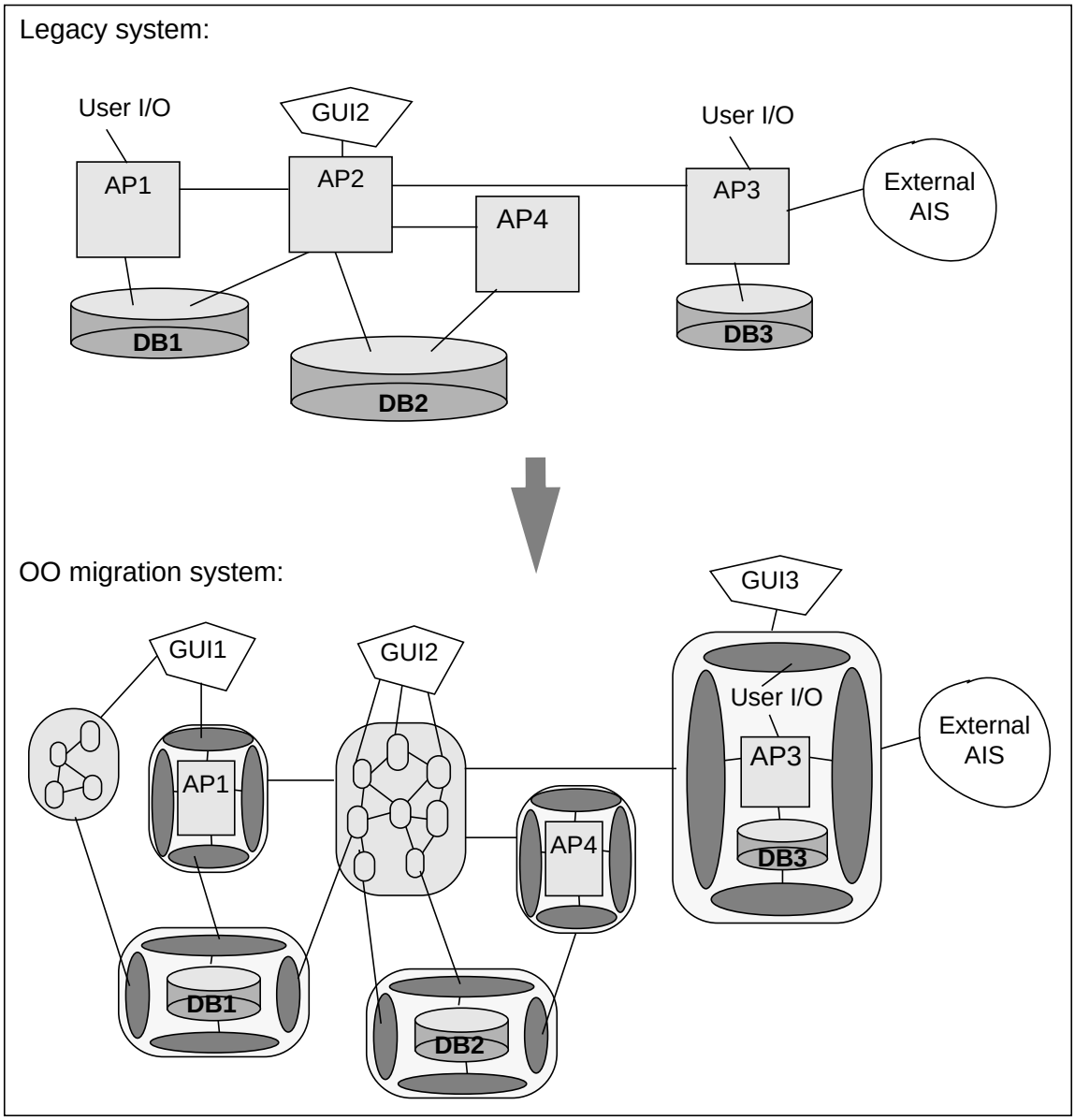

Figure 6. Divide-and-Wrap Migration Strategy

This strategy eases integration of all the pieces at different stages of transition since the methods interfacing wrapped objects can perform any necessary translations between legacy and modernized components. It offers flexibility in scheduling the transition increments through variations in both the amount of reengineering and the granularity of the components wrapped. In some cases, whole subsystems may be wrapped for a lower cost transition stage, while wrapping may be executed at a finer level during transition stages when more time and staff resources are available.

A migration system resulting from such wrapping and reengineering procedures can enjoy multiple immediate advantages over the legacy systems, such as the following:

- Conformance with standardized data definitions
- Improved interoperability with other AISs
- Greater ease of user operations through new GUIs
- More effective display of information through new GUIs
- Functional consolidation of multiple legacy systems in one migration system

It can also benefit in the long term from the following:

- Improved maintainability of reengineered code
- Lowered maintenance costs of fewer systems
- Eased restructuring of internal data through encapsulation
- Eased modification of legacy code through encapsulation
- Potential reuse of objects (reengineered or wrapped)

Software wrapping can thus ease the transition of legacy AISs to migration systems that are fewer, easier, and less costly to operate, and easier to modify, fully modernize, and maintain.

The principal drawback to this general transition strategy is that wrapping large portions of a system may require considerable rework (of object hierarchies, methods, and data structures) when these components are unwrapped and decomposed into more meaningful objects. Simply wrapping a whole program or database as an object cannot be expected to create objects that correspond well to the real-world entities that are the natural objects of interest in our information systems. When these wrapped objects are unwrapped and decomposed into constituent domain objects and methods at a later stage of migration, it

will be necessary to rewrite their previous access methods and all messages for them from other parts of the system. More extensive initial OO analysis can support an alternative strategy that promises to reduce both the revamping of object hierarchies and the rework of messages over the whole course of system migration, as discussed in the next section.

3.3 UNITE-AND-CONQUER

A "unite-and-conquer" strategy [TAYL92] achieves a unification of system applications and databases through a common OO framework that organizes access to legacy code and data as well as to new and reengineered OO system components. Such a framework can be constructed as part of developing business or enterprise models of the business activities supported by the legacy information system.[1] OO business models are portrayed in [TAYL92] as a sort of intermediate layer in a migration system that mediates between new applications and object data stored either in legacy databases or in new OO databases. Applications would send messages to business model objects requesting attribute data that the objects retrieve from the databases. Some existing applications might be replaced entirely by the business model or by applications built on top of it.

Legacy code can also be modified to access its data through the business models, thereby encapsulating the data and rendering the legacy code immune to disruption from modernizations of the data stores. This conception of unite-and-conquer is illustrated in Figure 7 on page 20. The business models are illustrated by an oblong region containing the classes/objects of the models and their interconnections. Illustration of the connections from modernized code to the business model and from the business model to legacy databases is simplified with single lines representing the multiple connections to and from individual objects in the models. The reengineered version of application AP2 is shown with both direct and mediated access to the database, DB1, although even its direct access is mediated by its own objects, effectively maintaining the encapsulation of the data in DB1.

While none of the legacy data stores are altered in this example, it is quite consistent with this strategy to directly modify the data stores. Databases could be encapsulated as objects or upgraded to full OO databases. Further encapsulation of the databases in the example, however, would be of little or no advantage. Mediation of database access by the objects of the business model already encapsulates the databases at a finer granularity than simply wrapping them whole as objects. Upgrading the legacy databases to full OO data-

---

[1] See [TAYL92], Chapter 6, "Creating an Object-Oriented Information System," for more on business models.

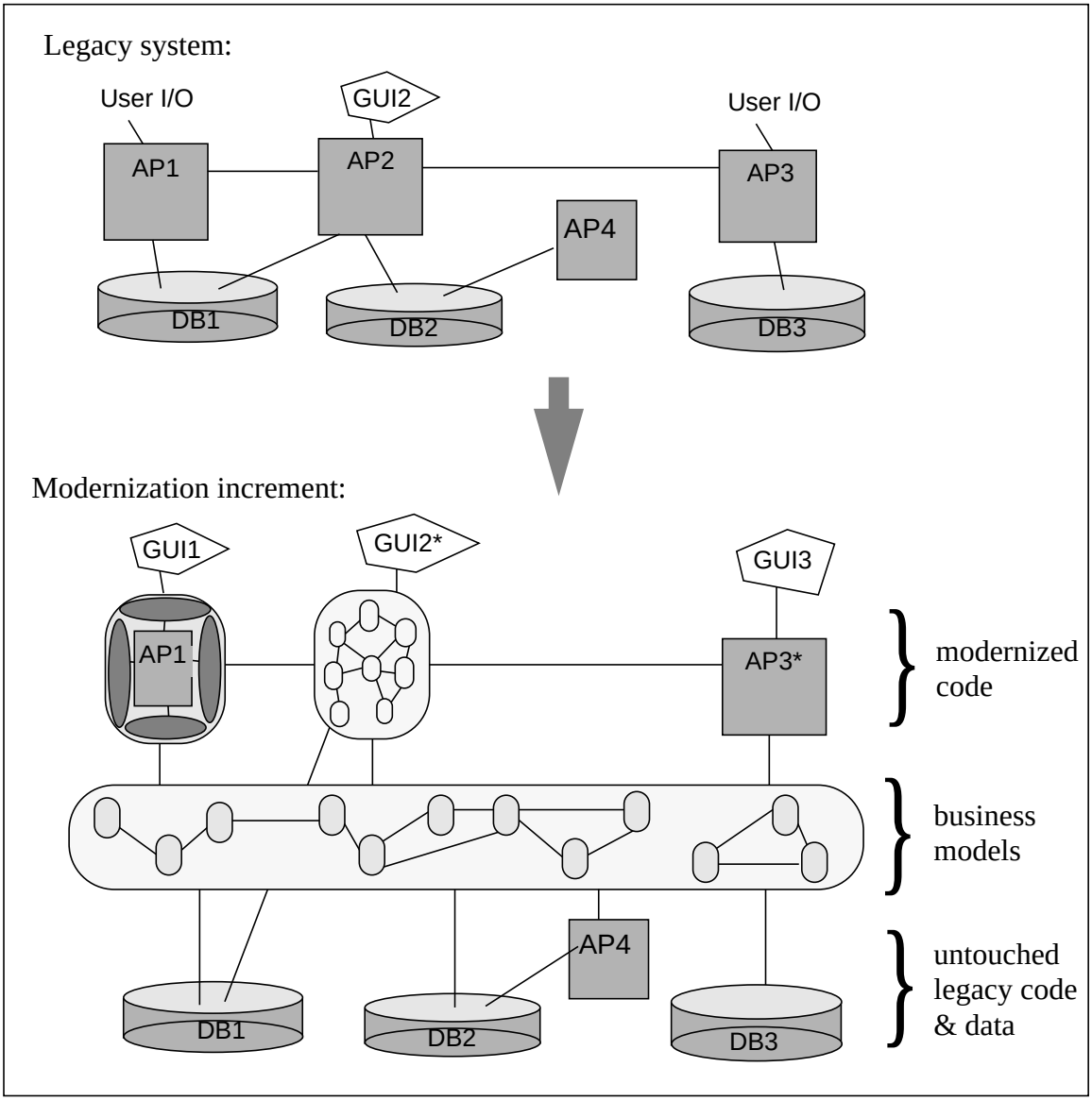

Figure 7. Unite-and-Conquer Strategy

bases may provide some advantages in terms of maintainability or in access efficiency for complex objects. However, it is quite feasible to retain a legacy relational database management system (DBMS) for the physical storage of the object data of the business models. Some object-oriented DBMSs actually use this approach to storing the attributes of persistent objects.

Legacy application programs may be transitioned to different levels of modernization during a unite-and-conquer migration stage. Figure 7 on page 20 illustrates direct wrapping of application AP1 augmented by a GUI, full OO reengineering for AP2, and

simple database access modernization for AP3. In this last case, the modernized program AP3*, illustrates another alternative for interfacing legacy code with encapsulated data: while the program is not object oriented, all calls from it for data access/update are all replaced by calls to objects in a business model, which then access the database. In fact, all of the first three applications access legacy data through classes/objects that effectively encapsulate the underlying databases. Application AP4, in contrast, is left unmodified in this example, so it does not make use of the business models and continues to access its data directly from DB2; hence DB2 is not fully encapsulated. A unite-and-conquer strategy offers many such choices of modernization alternatives, which can be tailored to fit the available resources and constraints at any particular stage of migration.

Unifying object models provides the unite-and-conquer strategy with substantive advantages over the previous strategies. These models provide transparent access to the data stores throughout the whole migration process. This supports incremental modernization of the legacy system while minimizing costly revisions to object models and data access code. Business models provide a new OO perspective on the business domain that can be helpful in guiding subsequent modernization phases. Business model objects should expect considerable reuse at subsequent phases of migration, and possibly even in other systems, thus lowering costs of subsequent migration activities. The business model also provides a core of system objects whose use can accelerate development of new applications, if they are needed.

However, developing business models can be a time-consuming analysis task for large systems since the essential business objects must be identified and mapped to the relevant existing programs and/or databases. Thus, a unite-and-conquer strategy can only be effectively executed at a migration stage when sufficient resources are available for this extensive analysis. When the resources are available, the payoff can be considerable in later stages of migration. When the resources are not available at a particular stage of migration, then a more piecemeal strategy can be adopted, such as divide-and-conquer or divide-and-wrap.

3.4 ONE-SHOT REBUILD

Thus far, all the OO migration strategies considered have been incremental. What about rebuilding an AIS in one round of the traditional analysis-design-implementation cycle? Should that be considered as a viable alternative to incremental migration? Ordinarily, no. Some experts [BROD93] refer to this strategy as “Cold Turkey,” and argue convinc-

ingly that it carries substantial risks of failure, at least for large, critical AISs. Another OO author and consultant [TAYL92] goes so far as to advise a business to declare Chapter 7 bankruptcy if taking a one-shot rebuilding approach, since he predicts both will lead to the same result. Multiple experiences in building large OO systems indicate that, for OO systems in particular, incremental development is more effective than the classic waterfall development model (as explained in the companion report on the OO development process [IDA95a]).

While a one-shot waterfall development has never been recommended for OO systems, it can be feasible in smaller AISs to apply locally incremental development to the system as a whole. Consider, for example, a small information system that is of comparable size to a single application in a larger system. There need not be much difference between the reengineering strategy for the small system and the large application. In small enough such systems, the OO analysis, design, and implementation might well proceed with respect to the system as a whole, without the need to defer treatment of any particular legacy components (applications or data stores). Such a unified rebuild would still best proceed incrementally, although the increments would be dictated by the OO analysis rather than by the legacy system components, and the reengineered system might only be scheduled for operations after the entire development was complete. When this sort of one-shot rebuilding of smaller legacy systems (or subsystems) is feasible, then interim measures, such as wrapping or gateways, may be unwarranted for that system, although they may still prove valuable in a broader context of other interacting systems.

One-shot rebuild could also be viable for a large legacy systems if it is very similar to an existing OO system that has already been implemented, or if it can be constructed out of existing tested frameworks and repository objects. In other words, if all the main pieces of a system have already been implemented in an OOT, it may be feasible to put them all together, tailored for context, to replace a legacy system in a single migration step. With the growth of object repositories and OO application frameworks, this method of system development may be expected to become more commonplace.

4. WRAPPER CONTENTS

Legacy software and data can be partitioned in many different ways to isolate those components that are most amenable to wrapping. Depending upon the criteria discussed in Section 2.5, analysis might indicate any of the following types of candidates for the contents of an OO software wrapper: function or procedure, data file, database, application program, or subsystem. Each of these different applications of wrapping is discussed individually in the following sections.

4.1 WRAPPING FUNCTIONS OR PROCEDURES

The lowest level of legacy code wrapping is realized when the contents of a wrapper is an individual procedure or function. This level of wrapping can be appropriate when parts of a legacy system are being reengineered using OOT, but some of its functions are difficult to rewrite in OO code, for reasons such as size, complexity, or absence of a required domain knowledge or expert.

Figure 8 on page 24 illustrates the idea of transitioning a legacy application program to an OO program in which some of the legacy code is retained beneath the wrappers of its objects. The two alternatives of direct wrapping and wrapping via domain objects are illustrated.

> *When feasible, it is best if the wrapper objects are abstracted from the application domain rather than directly from legacy software components since domain objects will integrate better with a domain model than will software objects.*

In both cases, only a single function (F4) is selected for wrapping, although many functions compose the legacy program. The other functions are presumed to be reengineered into operations of objects in the new OO migration program. More generally, any number of legacy functions may be selected for wrapping or reengineering depending on their suitability, as discussed previously in Section 2.5 on wrapping criteria.

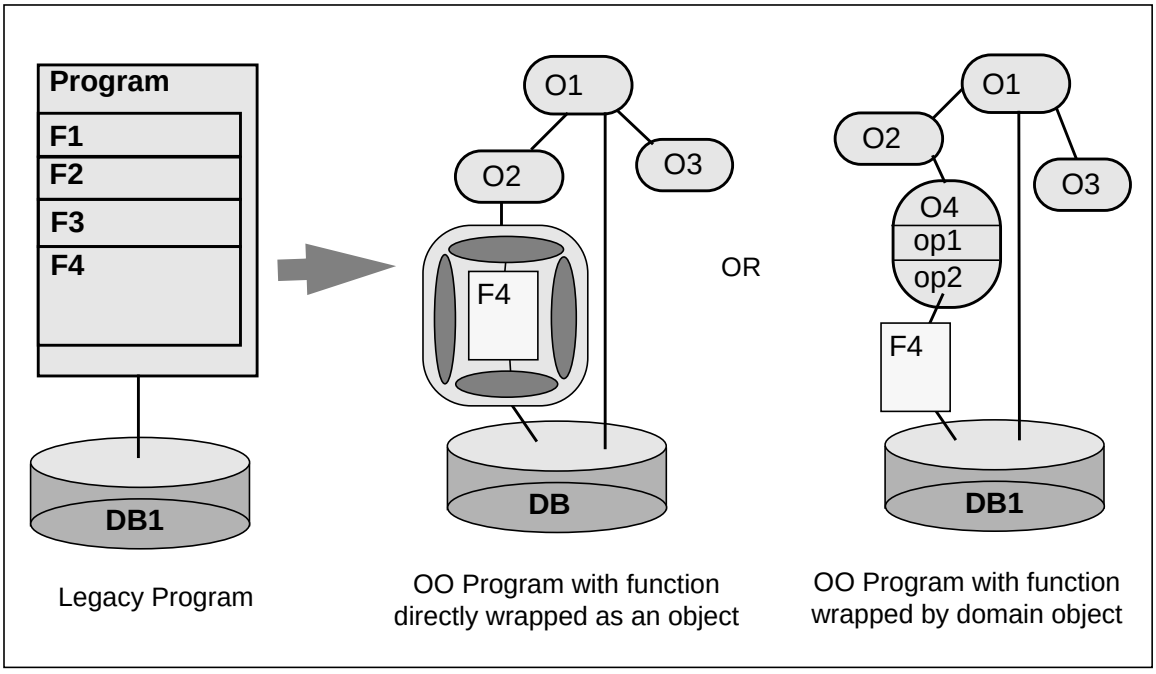

Figure 8. Wrapping Program Functions

In the first alternative of Figure 8, the function itself (F4) is transformed into an object/class in the new OO program, as indicated by the special-purpose wrapping icon. Such an object would be an instance/subclass of a class of abstract function/procedure objects. This is the simplest type of mapping from the legacy functions to objects/classes in an OO migration program, requiring the least amount of OO analysis of legacy software and requirements.

The second alternative uses a natural domain object or class (here labeled O4) from the application domain to wrap the function, which is then accessible only through this object/class. In this case, however, there is no need to "objectify" functions into a class of abstract objects; the function (F4) does not appear as an object in this system, but merely as external code accessed by an operation (op2) of a domain object (O4). This is a degenerate case of wrapping with a domain object model, in which the wrapping model consists of a single domain object. When wrapping is performed at the level of functions, it may be more natural to use a single type of object to wrap a given function, since functions often have a dominant association with one particular class of objects. In many cases, a suitable class to which the function applies can be derived from one of the arguments of the legacy function, which may themselves refer directly to objects or indirectly in virtue of standing for a property of a class of objects. When wrapping is performed at the level of programs or subsystems, it is more likely that a multitude of object classes will better represent the

object structure of the legacy code being wrapped.

> *Wrapping functions as operations of application domain objects ordinarily provides a better basis for subsequent migration to a fully reengineered system since legacy functions/procedures normally correspond better to methods than to domain objects/classes.*

Creating object wrappers at the level of functions and procedures offers potential benefits over coarser-grained wrapping at the level of whole programs or systems. Finer-grained objects can be reused without the encumbrance of ancillary code that may be irrelevant in other applications. It can also ease the transition to a fully reengineered migration system to have already decomposed programs into sets of interacting objects. These advantages of fine-grained domain object wrapping accrue at the expense of higher initial development costs compared to some of the coarser levels of direct wrapping. Choices of appropriate levels of wrapping are likely to be driven largely by the ease of decomposition of legacy software and the time and cost constraints at any given stage of software migration. With more time and resources, finer-grain wrapping is feasible, while tight time and resource constraints may require coarser-grain wrapping. Thus, the OO technique of software wrapping provides the migration team with considerable flexibility in meeting these constraints.

In some cases it may even be advantageous to wrap all, or most, of the procedures from a legacy program, rather than rewrite any of them initially. This could reduce the initial transition costs as compared to full reengineering, while providing a whole set of program or domain objects that conform to new data standards and might be reused elsewhere. Full modernization of the legacy code could then proceed in small increments, object by object, with minimal adjustments to the object structure only when indicated by deeper analysis.

4.2 WRAPPING DATABASE FILES

While the database of Figure 8 on page 24 is unchanged from the legacy system to the modernized one, it too could be wrapped to better encapsulate the data. Individual data files, data tables, and even whole DBMSs could be wrapped as objects. Figure 9 on page 26 illustrates two alternative ways to wrap an individual data file: directly as a file object, or as a domain class representing a set of domain objects whose data are contained in the file. Direct wrapping is represented using the standard direct wrapping icon introduced pre-

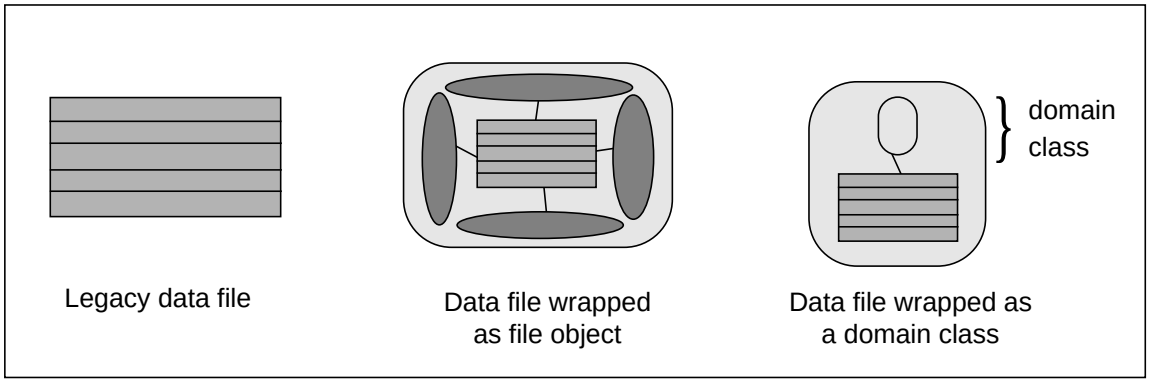

Figure 9. Wrapping Data Files

viously in Figure 1. Wrapping as a single domain class is the simplest case of object model wrapping (discussed in Section 2.2) in which the object model consists of a single class. A single legacy data file might also be wrapped with multiple domain classes if it contains data for multiple types of domain objects.

Systems with isolated data files independent of any DBMS might be transitioned to an OO system using either of these alternatives. Simply wrapping the data file with query, update, and delete methods could serve the purposes of some data standardization requirements and encapsulate the data to isolate its internal format from its access methods. A full transition to wrapping with domain classes might better support reuse and modularity, though it may incur additional costs from the additional restructuring when developing relevant object classes.

4.3 WRAPPING DATABASE TABLES

Separate database tables within a relational DBMS might also be effectively wrapped by restricting access to a set of methods defined for each table or group of tables, as illustrated in Figure 10. Conceptually, the server, or virtual copies of it, is effectively wrapped with each wrapped database table, since access to such data must be mediated through its server. Thus, in order to conform with the OO paradigm of encapsulated data, access to the server would have to be restricted to the developed object methods (either by design conventions or system constraints). So the server is effectively encapsulated in order to encapsulate the data it serves. This approach to wrapping data can offer some of the benefits of objectifying a database without performing a full decomposition into primitive objects.

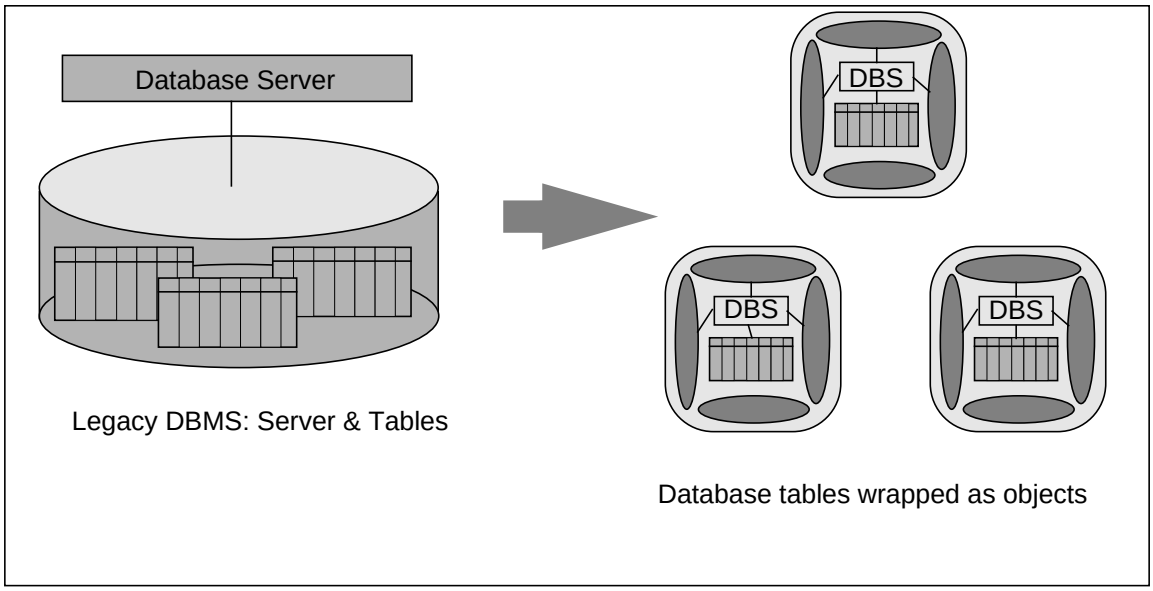

Figure 10. Wrapping Database Tables

A more natural approach to wrapping data within the OO paradigm breaks up tables into multiple objects at a finer granularity. If a database table can be identified with a primary key, such as that of employee in a table of employee records, then it may easily map into a class corresponding to the general type of that key, e.g., of employees, with each row in the table corresponding to an object instance of that class. The primary key values then name the individual objects, while other table entries correspond the attributes of these objects. This approach is illustrated in Figure 11, where the OO wrapping of each database

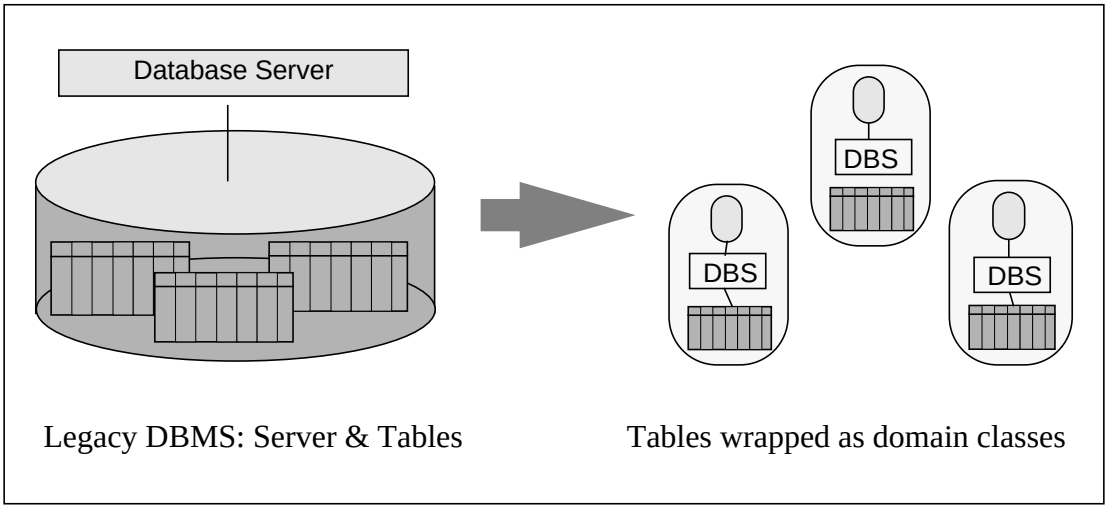

Figure 11. Wrapping Database Tables as Domain Object Classes

table is accomplished by the class that interfaces to it. Hence, the domain class is drawn as constituting part of the wrapper. Attributes of a class instance are retrieved by queries to tables keyed on the objects of that class. Unlike directly wrapped tables, such domain object classes may incorporate attributes and methods beyond those required to represent and access table values. As with other wrappers that create meaningful domain objects, this approach has the advantages of better integration with an OO system, although it may involve more work in abstracting the appropriate classes and building the OO interface. This is the standard procedure for converting database tables to objects when a legacy relational DBMS is reengineered using OOT.

Such a transition may fall short of a full reengineering in several respects. The database may not be fully restructured to support access between associated objects based upon associations of a full object model. The formats and contents of legacy database tables may be retained even though they do not map one-to-one to domain object classes. When such a mapping does not exist, the domain object wrapping of the database tables is better viewed as a wrapping of a set of tables (or entire DBMS) by an object model, as illustrated in Figure 12.

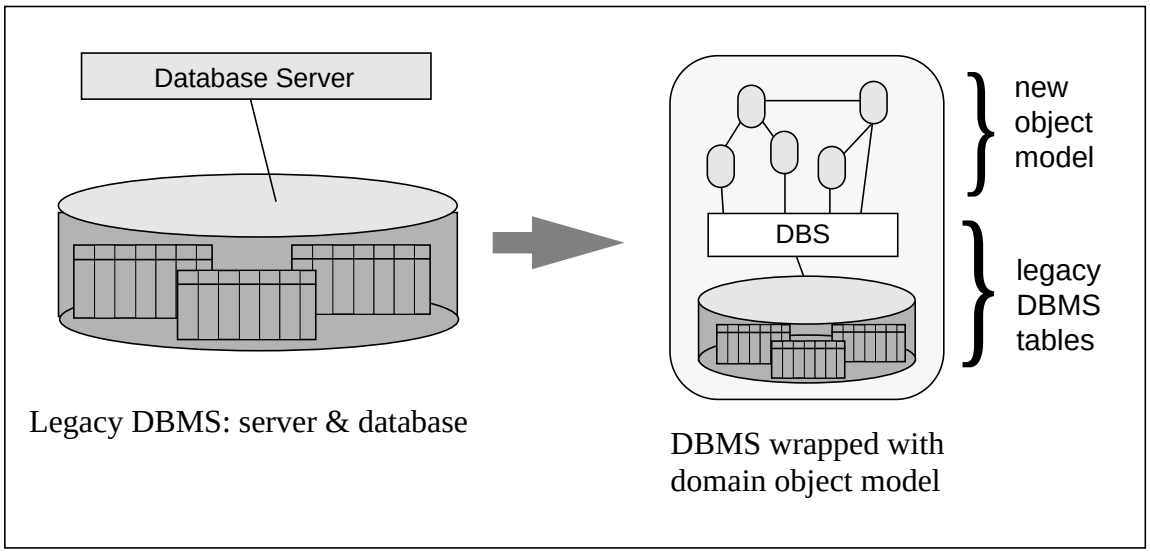

Figure 12. Wrapping Database Tables with a Domain Model

4.4 WRAPPING A DATABASE MANAGEMENT SYSTEM

One potential drawback in separating data tables into distinct objects is that query response time may be adversely affected when table joins are involved since a join must be reformulated as multiple queries in order to access multiple tables via their separate encapsulating methods. A simpler alternative that might minimize this problem is encapsulating

the entire database as an object, as illustrated in Figure 13. This approach may retain the same order of efficiency as queries in the legacy database because the same database queries could ultimately be invoked by the database object's methods. Complex queries could be posed directly to the wrapped DBMS object and processed as joins. The only extra query costs are the small constant-time overhead incurred passing through the methods. The data are all encapsulated, so that the access methods can be independent of the data storage organization. Wrapping a whole database system like this can be an efficient means of establishing compatibility with new interoperable data standards while providing a framework for transparent incremental modernization of the internal data representation. However, this approach might have to compromise some of the modularity and encapsulation implicit in an object model of the database since access to stored objects could not all be mediated by their distinct classes if complex queries are to be handled directly. Access to objects cannot always be mediated by the operations of their individual classes if complex queries about different classes of objects can be sent directly to such a DBMS object wrapper.

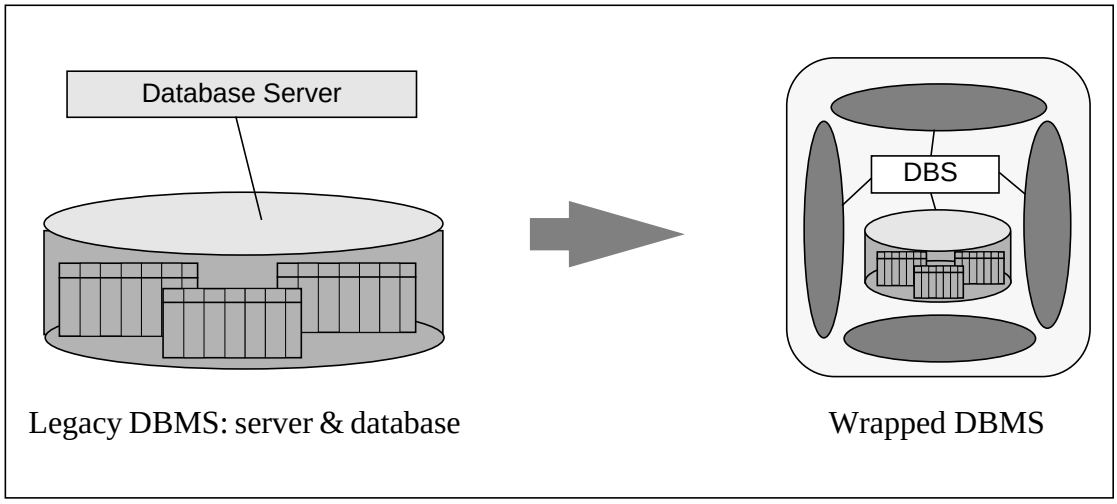

Figure 13. Wrapping a Whole DBMS

4.5 ALTERNATIVE DATABASE ENCAPSULATION MODELS

Object wrapping is not the only or perhaps the most natural means of achieving encapsulation of a legacy database. An interface or "gateway" to a database can be written to hide its internal structure[1] without conceptualizing the result as an object or set of objects. Such alternatives achieve very much the same effects as direct wrapping when applied to databases, since a database wrapped as a single object is unlikely to participate

[1] Software gateways, as presented in [BROD93], are discussed in Section 5.1.

in any of the other distinguishing features of objects. Wrapping an entire database creates an object too large and unstructured to be a likely candidate for hierarchies, inheritance, or reuse. It is not that databases cannot be decomposed into useful hierarchies of objects, but that wrapping a whole database does not provide any such decomposition. Thus, casting an encapsulated database as an object offers little advantage over "gateway" conceptions of encapsulation. While wrapping does provide some uniformity within an OO system, the principal benefit of information hiding derives from the encapsulation, so that other methods of encapsulation might do as well.

One interesting proposal using object models for database encapsulation is the "three-tier solution" for the problem of maintaining database integrity within a distributed information system with heterogeneous databases [LOO94]. This solution involves using a set of OO database servers as a middle tier between presentation/application software and the distributed heterogeneous databases of large information systems. This middle tier would have a central OO object model—a global conceptual schema—that defines a common global view of shared subsets of all the local conceptual schemas of the databases it accesses. Such an object model not only encapsulates the data of all the underlying databases, but also functions to enforce system-wide data integrity strengths. Such system-wide constraints could not all be enforced locally by bottom-tier database servers because they do not have access to all of the relevant data or knowledge of the relevant applications. The data encapsulation provided by this scheme can also support incremental transparent modernization of data stores. This proposal is one more example of how object models can be useful for wrapping data stores within large information systems.

4.6 WRAPPING PROGRAMS

Entire programs can be wrapped just as well as procedures and data, although special care may be involved in mapping their I/O interfaces into the OO paradigm. The interfaces of wrapped procedures or functions are generally straightforward to adapt: simply replace calls to them by calls (or messages) to their encapsulating methods. Interfaces to wrapped data files or databases are simply reconstructed largely by replacing direct queries and updates with calls to methods that generate them. The interface structures of legacy programs, in contrast, are typically complicated by their bi-directional interfaces to users, via keyboard and monitor I/O, or a GUI in more contemporary systems. When the user interface exchanges I/O directly with an external device (or device buffers), such as a keyboard and monitor, its treatment within an OO system admits many alternatives. I/O devices themselves could be remodeled as objects which engage in message traffic with the

wrapped program object; the whole interface could be reworked as a GUI in which GUI elements are modeled as objects; or the user interface could be left alone with some violation of the OO tenet of encapsulation. Some of these alternatives are illustrated in Figure 14 on page 31.

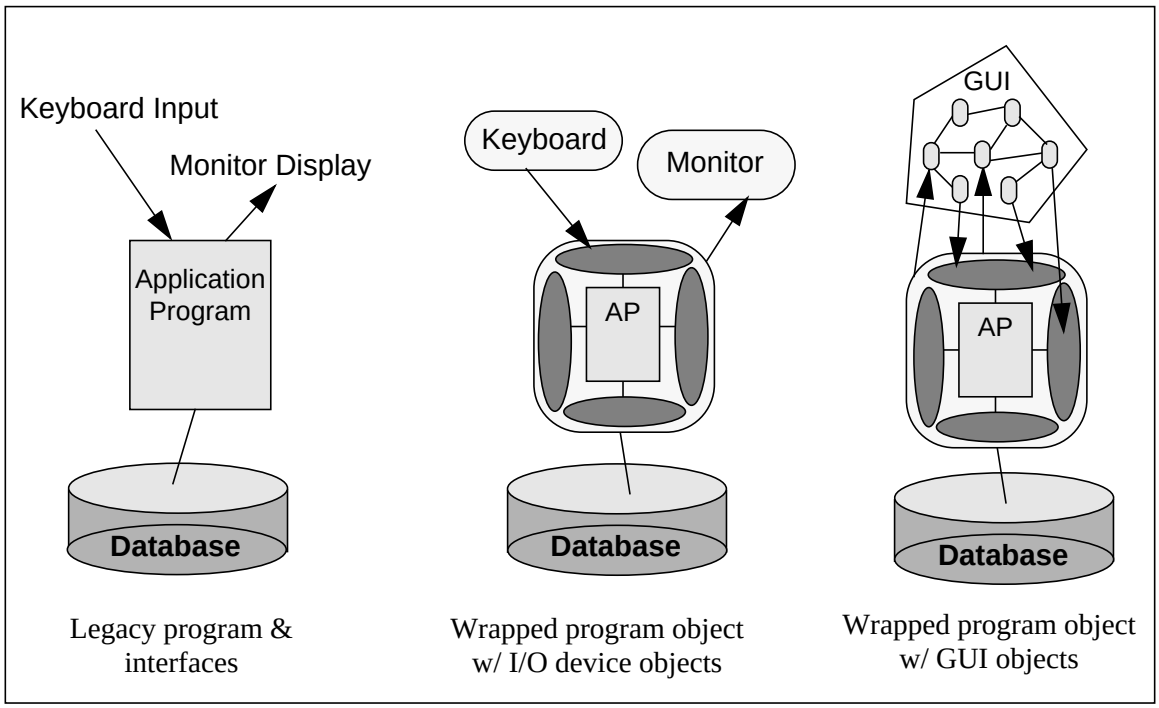

Figure 14. Wrapping Programs as Objects

Another class of alternatives is generated from the technique of wrapping with an object model constituted of multiple objects. This approach generates object classes and their instances from analysis of the application domain, interfacing these objects to the legacy application program via either an API or direct calls to the legacy functions or subprograms, as illustrated in Figure 15 on page 32. Such object model wrapping provides finer granularity in the generated objects and holds more promise for abstracting objects/classes of lasting value throughout subsequent software migration phases (if any).

If I/O devices are themselves wrapped into objects, then input device objects provide program input by sending messages to the program's methods. A program's output would then direct output by sending messages that request output services to the methods of the output device objects. Thus, this approach would require modifying the output code of the program to call the appropriate display device methods. If user I/O were reworked using a GUI, then individual GUI objects would communicate with the program object via messages, assuming an object-oriented GUI.

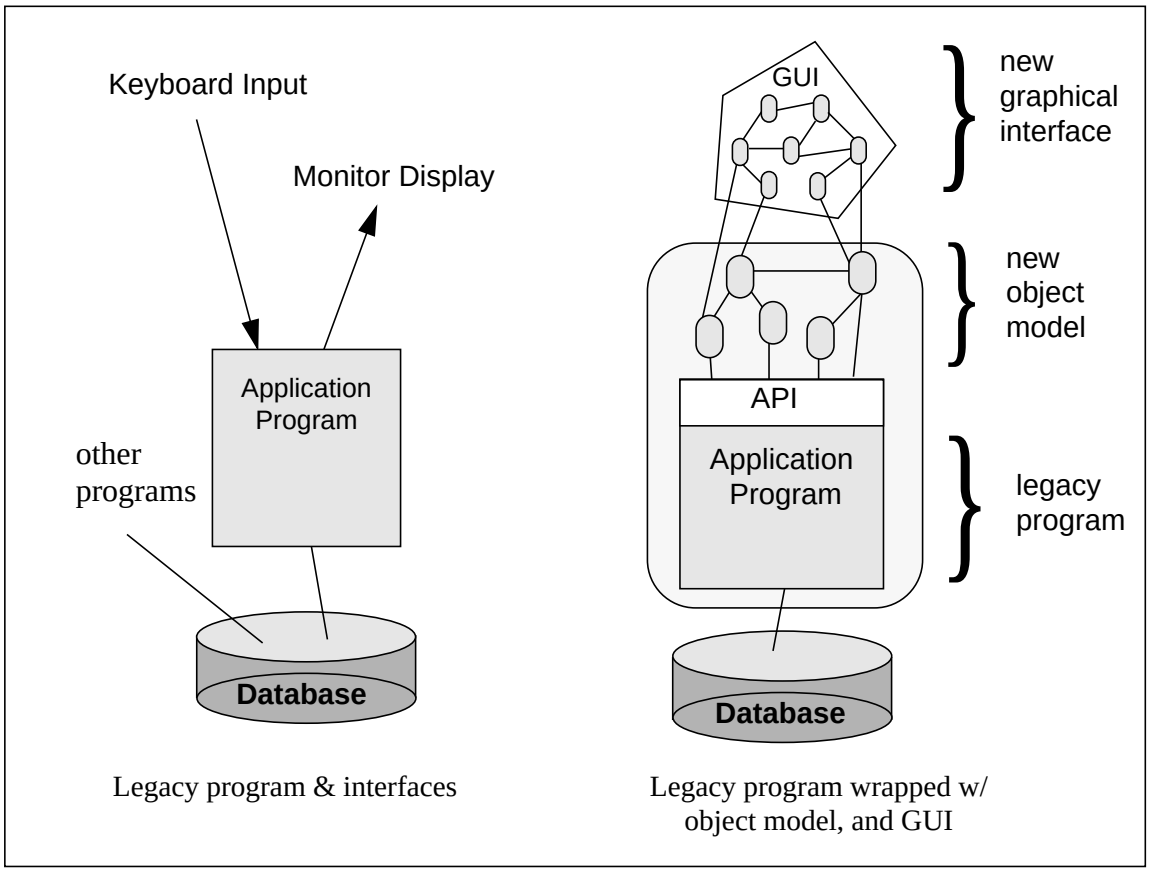

Figure 15. Wrapping Programs with Object Models

Creating a GUI for program I/O, while obviously a more costly alternative, can provide some unique advantages in the larger context of software migration. In isolating the I/O from the program, it supports the incremental modification of either one independently of the other. Monitor displays can be reformatted to take advantage of graphical display techniques, and programming functionality may be added through the user interface without disrupting the wrapped legacy programs. This latter capability is of special value when multiple software applications are being migrated to a single encompassing application. In such cases, which are expected to dominate the DoD software migration program, a single existing system may be selected as a basis for the targeted migration system. These selected legacy systems cannot always be expected to include all of the functionality of the deselected systems. In many cases, a legacy system selected for a migration system target will require additional functionality in order to fully meet the requirements of all the legacy systems being replaced. In such circumstances, the separation of program from user interface can provide the framework for augmenting the selected legacy system with minimal disruption to the original code. The combination of wrapping legacy programs and inter-

facing them to a GUI may thus provide the most effective framework for a staged migration of legacy systems in many such migration contexts.

4.7 WRAPPING SUBSYSTEMS

The next level of complexity of wrapped software/data objects is wrapping subsystems where the scope of the wrapper is expanded to include one or more data stores along with a program, or programs, as illustrated in Figure 16. The user interface for such wrappers admits of the same variety of treatments as do the wrapped programs just discussed; I/O may use the legacy procedures filtered through the wrapper interface, or a separate object-oriented GUI may be created, as illustrated. The separation of the user interface into a GUI, even a primitive one, can be advantageous for independent modification of interface and program functionality, as discussed previously.

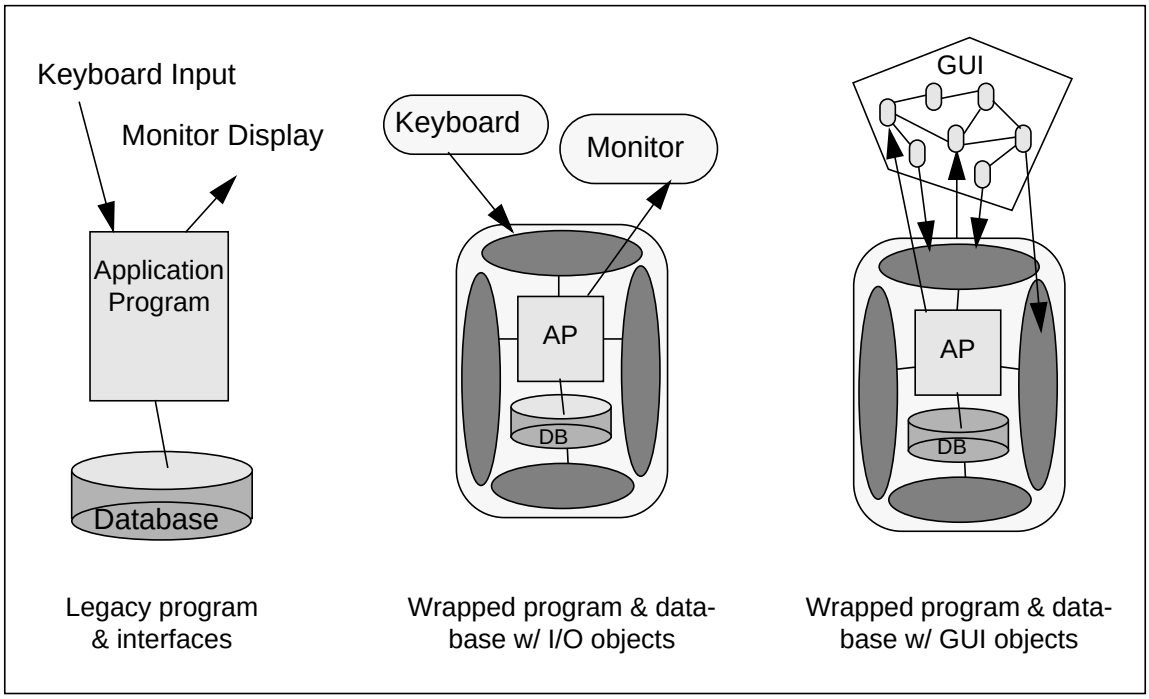

Figure 16. Wrapping Program and Data Stores as an Object

This level of wrapping is least disruptive when the data stores wrapped with the program are used exclusively by them. Otherwise, the encapsulation of the data created by wrapping it with the rest of the subsystem would require all other access to this data to be mediated by the wrapper. Thus, any external access routines for such a wrapped database would have to be rewritten to access it via the wrapper.

The most complex of wrapped software objects can encapsulate entire software systems or subsystems, including a user interface along with application programs, and databases. Such a software subsystem might be selected for wrapping if it was too challenging to decompose at a given stage of migration but needed a coherent interface with other components of the larger system. Figure 17 illustrates the transition from a legacy system to a migration system in which an entire subsystem, consisting of database DB3, application AP3, and its graphical user interface GUI3, is wrapped as an object. The database, DB3, wrapped in with this subsystem is uniquely accessed by the associated program, AP3, so that wrapping does not require any changes to its access. In this example, the new interface GUI3* has been added outside the wrapper of the subsystem in order to establish uniformity of user interface with a new system-wide standard.

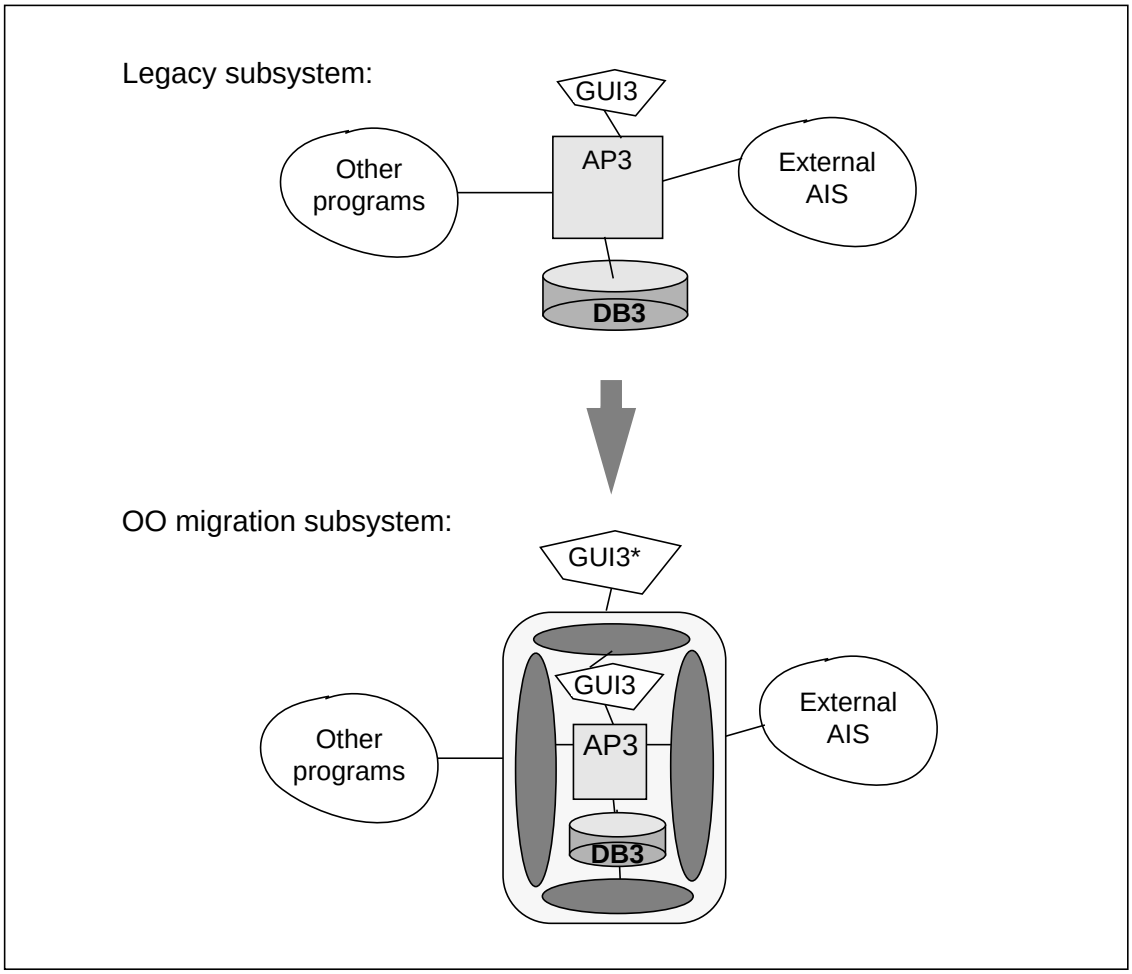

Figure 17. Wrapping Entire Subsystems

5. ALTERNATIVE ENCAPSULATION TECHNIQUES

Much of the benefit of the wrapping technique derives from its encapsulation of wrapped legacy software or data. OO wrappers, however, are only one means of achieving encapsulation. To provide a broader perspective on encapsulation, we discuss some alternative conceptions of encapsulation for legacy systems. These conceptions may also contribute to AIS migration strategies whose goals include incorporation of OOT, although they need not be object oriented in themselves.

5.1 GATEWAYS

Gateways are described by Brodie and Stonebraker [BROD93] as software modules placed between operational software components that control communications between them. Gateways are discussed as the main device of their methods for incremental migration of legacy information systems. Gateways can insulate software components on one side from changes made to legacy components on the other side, thereby supporting incremental modification of legacy components without disrupting the rest of an information system. The three types of gateways in Figure 18 on page 36 are distinguished by their placement within an information system: (1) a database gateway between a database and an application; (2) an application gateway between an application and a user interface; and (3) an information system gateway between a whole information system and the users (and any interacting AISs).

Gateways are very general types of software mediators: they can achieve information hiding in multiple directions, they can be formulated within any type of language or programming paradigm, and they can mediate between many different types of software components, e.g., conventional programs, modules, functions, databases, interfaces, and users, as well as objects. In this respect, a gateway can be seen as a generalization of a wrapper, as a wrapper establishes a one-way gateway through its methods into a software component in creating an object. The wrapping of methods around a legacy software component can be considered a type of gateway to that component. But gateways need not be restricted to containing a single object. The OO implementation of an entire business

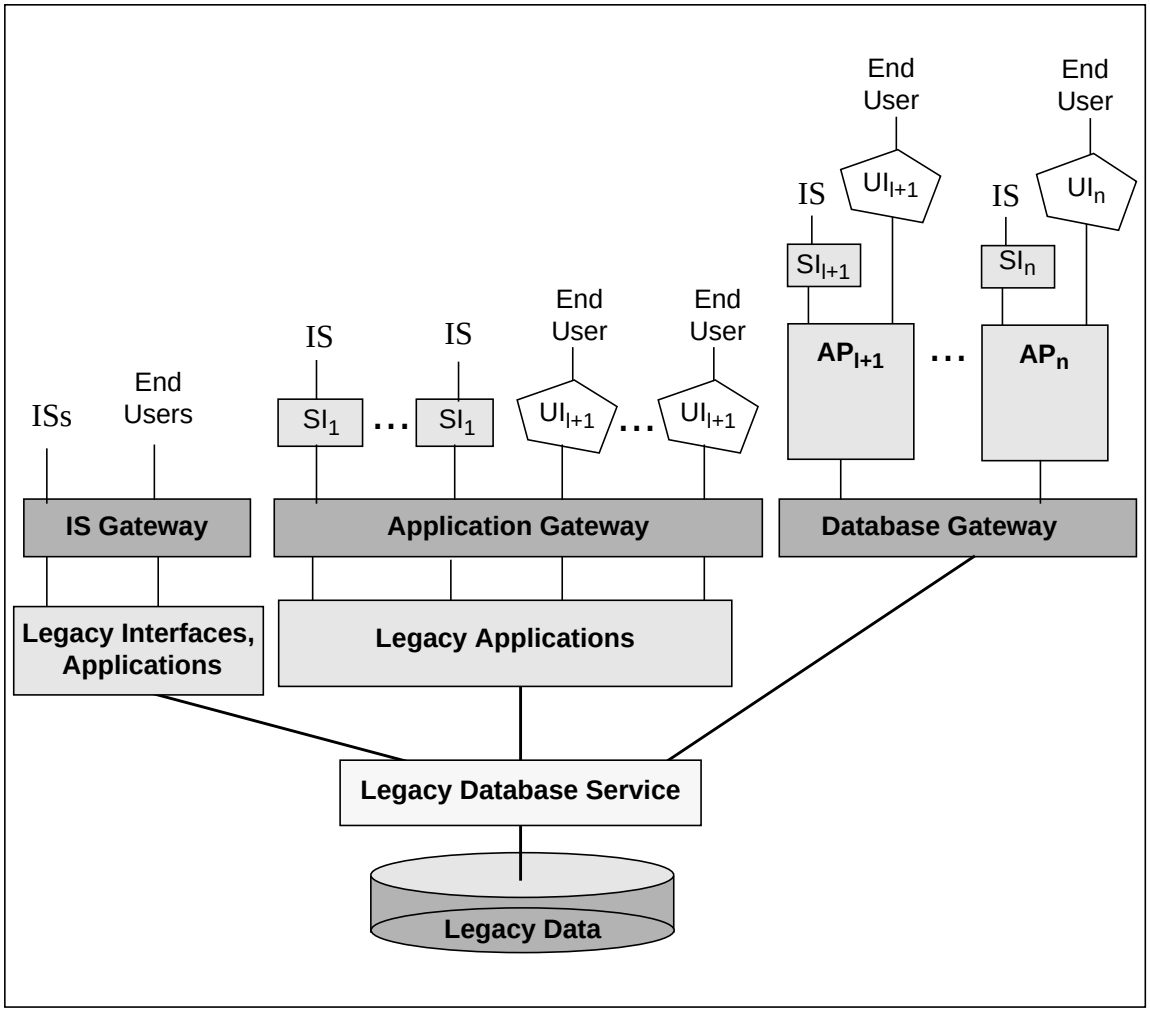

Figure 18. Gateway Types and Placements

model, as illustrated for the unite-and-conquer strategy of Figure 7 on page 20, can be viewed as a type of gateway, in this case a database gateway between the modernized code and the legacy databases. A gateway with this type of structure, however, does much more than simply encapsulate the underlying legacy data: it provides a structure for new applications and for transferring the functionality of legacy programs to modernized software (e.g., business objects).

Even gateways without any sort of object orientation might be used effectively as part of a migration to an OO system. Gateways might be written in a non-OO language in early stages of a migration project because of the greater familiarity and confidence with that language by the available software engineers. Such non-OO gateways could still support transparent incremental migration of legacy programs and data to OO systems by isolating different levels of the system from each other. Eventually, any such gateway would

have to evolve towards an OO model in order to support communication with objects on different sides of the gateway.

Gateways may, of course, also be used for migration of legacy systems to modernized system without any object orientation. For such projects, the gateway's isolation of software components from changes can support an orderly incremental transition to the goal system, whatever software paradigm is used. Much greater depth on alternative migration methods utilizing gateways can be found in [BROD93].

5.2 DATABASE VIEWS

One very common method of data format hiding that deserves some mention is provided by the alternative views of data supported by relational database servers. A relational database typically provides different views of its data to different sets of users in order to support access security constraints, as well as to provide convenient organization of output data. These views can hide the underlying logical database model; a user's view of a single table schema, for example, can hide a logical model composed of multiple relations. Thus, such views may be said to encapsulate the underlying logical structures, which may change while the views are unchanged. However, this sort of encapsulation is quite limited within an ordinary relational DBMS. Typically, user views are restricted to tables of data elements, and the data elements themselves cannot have much structure, being restricted to standard database types.

Thus, relational DBMS views do not ordinarily support information hiding or encapsulation to the full extent found in OO systems. View mechanisms may still be useful for providing a perspective on relational tables that makes them look like object attributes, so that a relational DBMS may be used to store attributes of persistent objects. But the relational DBMS itself will not support the polymorphism or full data hiding capabilities of an OO wrapper, a gateway, or an object model. Furthermore, a relational DBMS is, by design, limited to accessing data from its own data stores. So, it does not provide the type of encapsulation that will support transparent transition to genuine OO databases. Thus, the view mechanism of a relational DBMS by itself does not, in general, support the type of data encapsulation that will ease the migration of legacy databases. OO wrappers around relational DBMS tables, in contrast, support robust encapsulation of data, as discussed previously and in greater depth in Chapter 6.

6. WRAPPING IMPLEMENTATION

In this chapter, we elaborate on wrapping strategies using examples, and develop guidelines on how they might be implemented in Ada. The examples presented here illustrate the specific code-level details for a specific system environment. For similar environments, these example codes may serve as templates since all code has been tested and verified. In other cases, the examples illustrate a variety of code-level techniques for wrapping which may be applicable depending on the environment.

Since Ada 95 does not have validated compilers at the time of this writing, all OO migration examples here are implemented in Ada 83 which has readily available validated compilers. Due to the general upward compatibility, most of the example code should execute under any validated Ada 95 compiler. Because Ada 83 is not a fully object-oriented language but only object based, certain aspects of the OO features of inheritance and polymorphism have to be explicitly coded. The various alternatives for coding OO features in Ada 83 are explained in [IDA95b]. Most of the legacy application examples used here are based on Cobol, the dominant information processing language of the past several decades. Examples using Fortran, C, and Assembler are also included because they too have been used in implementing legacy systems.

This chapter begins by examining various constraints that existing DoD legacy systems may impose on migration systems when attempts are made to retain some parts of them. Then the basic prerequisites for any particular application of wrapping technology are reviewed. Program functions are the first class of software components whose wrapping implementation is described. Several examples of wrapping using Ada interface pragmas are given for functions or subprograms written in the Cobol, C, and Fortran languages. Next, a simplified scenario is presented of a legacy migration situation as a basis for illustrating wrapping techniques. Details are provided on wrapping a data file and a program from the legacy system using Ada interface pragmas. Complete code for this example is provided for reference in Appendix A.

When interface pragmas from Ada to a legacy programming language are not supported by the migration environment, alternative techniques exist to support this interface. Several such techniques are described: use of common areas to exchange information, calls through operating system services, and calls through intermediate languages. Next, the details of wrapping databases, focusing on the interface between Ada and SQL are discussed. Finally, we describe the basics of the new language bindings in Ada 95 which greatly simplify the interfaces to foreign languages, easing the implementation of wrapping in Ada.

6.1 LEGACY ENVIRONMENT CONSTRAINTS

Large portions of DoD legacy systems include obsolete hardware, technology, and systems which were designed almost 30 years ago and are often poorly documented. A sample of several legacy information systems are listed in Table 1. Many of these decades-

Table 1. Examples of Legacy Environments

System	Languages	Data Handling Systems	Operational Environment
Defense Civilian Personnel Data System	Burroughs assembly and home-grown procedural language: Samuel	Home-grown database management system	Multiple sites using remote access
Defense Civilian Pay System	Cobol	IDMS/R	
Marine Corps Total Force System	Cobol and Assembly	VSAM and Adabase	
Composite Health Care System	MUMPS	Fileman	
Medical Performance Factors	MUMPS	Fileman	

old designs are pushing the limits of their engineered capabilities and, as such, cannot be readily adapted to open architectures and current technologies. For example, many of the systems use memory overlays managed by the application program. Although innovative at the time, such memory management techniques make it very difficult to adapt to new architectures and technologies since many contemporary system architectures do not support the older memory overlay technique. Such constraints in legacy systems create special challenges when migrating them to new platforms without massive reengineering. The soft-

ware wrapping technology may offer cost-effective solutions during rapid turnaround modernization for some legacy systems. Other legacy systems, however, may be too enmeshed in obsolete technology for wrapping within modernized software and hardware environments.

Decisions about whether and how to wrap various components of a legacy system will depend on the system-specific situation in addition to general criteria outlined previously. For example, legacy systems written in Cobol for IBM hardware and networking relied heavily on the Customer Information Control System (CICS) communication package which is embedded in the operating system. In such an environment, Cobol applications make calls to the operating system and supply the pointer for the data structure to handle file system and/or terminal I/O. The CICS intern communicates with the hardware and I/O managing the terminal and file system. Such operating system dependencies are found to be common in many legacy systems. When present, these dependencies can pose difficulties during porting one or more parts of a legacy system to current generation platforms. As a result, all such operating system dependent calls may have to be rewritten in order to migrate a legacy system to a new platform.

Vendors like the Digital Equipment Corporation (DEC) and IBM often provided extensions to high-level languages (e.g., Cobol, PL/1) to facilitate task and program management, terminal handling, database access, and I/O handling. Legacy applications traditionally relied heavily on these extensions to achieve high performance since many of these extensions are not directly supported on migration platforms and may create potential barriers to moving components of legacy systems. When a legacy system's Cobol extensions are not supported on a migration system platform, the portions of code that use the old extensions will have to be re-implemented before the legacy code can run under the migration environment.

When legacy systems are tightly coupled with the hardware, operating system, communication system, terminal handlers, etc., of an obsolete legacy environment, it may be best not to wrap their components within a modernized hardware-software environment. In such cases, a client-server strategy should be applied if a suitable interface can be established between the legacy system and the migration system. Under this approach, a legacy system (or components thereof) could operate as a stand-alone client-server and interact with other client applications using a messaging system.

An alternative approach can be based on identifying components of the legacy system that had little to moderately complex system-specific barriers to wrapping and salvag-

ing them for use as wrapped components in a modernized OO system. In this case, the wrapped legacy functions and the new system will coexist in the same computer system. Naturally, when a legacy system contains substantial system-specific barriers to wrapping, this will result in less salvageable legacy code.

If a translator can be developed that flags system-specific portions of legacy code and maps them into their equivalents in the migration system, much legacy code can be salvaged and the development effort can be accelerated. In some cases, a fairly uniform and semi-automatic translation procedure may be possible for porting code from a legacy environment to a migration target environment. Currently, several domain-specific commercial products are available to facilitate such porting.

The best approach for handling system-specific barriers to wrapping will obviously depend on the specifics of the legacy system and its intended migration environment. Alternative approaches should be analyzed relative to the resources, costs, quality, interoperability, and migration deadlines for a particular migration system.

6.2 WRAPPING PRELIMINARIES

Wrapping is a special activity of those software migration projects that incorporate object wrapping technology. Thus, its success depends on the execution of several preliminary stages of software engineering. A number of activities should be performed before proceeding to the details of wrapping implementation:

- Perform functional process improvement (DoD Directive 8020.1).
- Select system migration strategy.
- Perform OO analysis and OO design, including object modeling.
- Select levels of abstraction for wrapping.
- Select specific wrapping candidates.
- Select OO programming strategy.

DoD policy [DOD92] requires that a functional process improvement exercise be conducted before the initiation of all systems reengineering activities. Since migration of a legacy system may involve reengineering in addition to wrapping, one may have to perform a *functional process improvement* exercise as well before finalizing selection of the candidate legacy components for wrapping. The results of functional process improvement may affect the suitability of a legacy component wrapping candidate, based on the findings of

the exercise of its obsolescence or need for modifications. In general, legacy systems that require extensive modifications are ordinarily poor candidates for wrapping, since modification of such existing code and/or data might be just as costly or more so than a complete reengineering of the system.

A system migration strategy determines the general guidelines for migration, including the number of transition stages and the relative proportions of different transition modes applied at each stage. Software wrapping is one of those transition modes, along with reengineering, code modernization, and the simple retention of a legacy system component. Alternative system migration strategies utilizing different mixes of transition modes (as discussed previously in Chapter 3) are divide-and-conquer, divide-and-wrap, unite-and-conquer, and one-shot rebuild.

Except for one-shot rebuild, all are incremental strategies, allowing modernization to proceed in stages in which more legacy components are modernized at each stage. Wrapping may be used as a part of any incremental strategy to encapsulate a legacy component for ease of integration with the modernized components. Selection of a migration strategy begins the determination of the extent of application of wrapping in the migration process. Strategies like divide-and-wrap, for example, are committed to wrapping all legacy components that are not reengineered at a particular stage. The migration strategy should also indicate the type of wrapping planned, whether it is wrapping components as objects or wrapping them as domain object models, as explained previously in Chapter 4.

OO analysis and design result in an object model that defines the intended object's structure for implementation. This may involve OO analysis aimed towards process improvement requirements, as well as reverse engineering of the legacy application (as discussed in Chapter 5). The object model may be developed to different stages of completion during different phases of system migration based on the system's migration strategy. A strategy that commits to wrapping with object models may result in a complete object model for the entire domain during the migration phase. Modifications to this initial object model are to be expected during the subsequent stages when previously wrapped components are reengineered using OO technology. This is typical of the successive refinement of object models during any incremental OO software development. Performing the initial OO analysis for the entire domain may ease the transition from one stage to the next and minimize the disruption to the initial model. Less comprehensive OO analysis in the initial transition stage can lower the costs of that transition at the expense of greater overall costs due to disruptions during the later stages. In any case, the extent of OO analysis performed prior

to wrapping can greatly affect the structure of wrappers created and the ease of subsequent reengineering and integration of the content in those wrappers.

Two principal criteria must be considered in order to identify the candidates for wrapping: Is the wrapping feasible and is it justified? Feasibility is determined largely by the modularity of the candidate component and its support within the migration environment, as discussed previously in Chapter 4. Justification for wrapping, as contrasted with reengineering a legacy component or leaving it untouched, depends on a wide variety of factors, including quality of documentation on legacy component, departure of domain experts, complexity or fragility of code, size of code, resource limitations, expiring hardware and software contracts, and interoperability requirements.

Once a candidate for wrapping has been identified, the appropriate level of abstraction for wrapping must be determined: a subsystem might be wrapped as a single component or wrapped as many separate parts; a program might be wrapped as a whole or decomposed into multiple functions or procedures for separate wrapping; or a database might be wrapped as a whole or its tables or files may be wrapped individually. Table 2, "Wrapping Guidelines," on page 45, provides some general guidelines on wrapping a legacy system at various level of abstraction. Decomposition of components and wrapping at lower levels of abstraction offers the benefits of modularity though at the costs of higher wrapping transition overhead. Wrapping many small functions, for example, could take more time and effort than simply reengineering them for the new environment—therefore, this approach is not recommended. More generally, the costs and benefits must be weighed in determining a reasonable level of abstraction for wrapping each wrapping candidate. In some cases, candidates may be rejected after all of the costs have been fully accounted for.

Actual implementation of wrapping requires selection of an OO programming strategy, including the programming language and the OO style guidelines for implementing classes and objects within it. Although Ada 83 supports encapsulation, the support for inheritance is limited, and the support for polymorphism is deficient compared to most object-oriented languages like C++ or Smalltalk. Thus, fully OO programming within Ada 83 requires selection of implementation strategies for inheritance and polymorphism. It is preferred for its high levels of encapsulation and modularity. Its main weakness lies in the use of unchecked type conversion to extend class attributes in subclasses. The viability of this technique depends on the Ada compiler implementation of data record modeling and access types. However, this implementation dependency is not a problem for the majority

of Ada 83 implementations, and such code can ordinarily be ported between compilers without modification.

Table 2. Wrapping Guidelines

Level of Abstraction	Wrapping Overhead - Comments
Function	Overhead is very high. Not recommended unless the function is unusually complex & difficult to re-code.
Procedure	Overhead is high. Recommended but should initiate a performance evaluation before implementation.
Sub-Module	Low overhead. Recommended.
Module	Overhead is moderate. Should consider the execution environment.
Complete System	Recommended. Should look at client-server model during implementation.
Data File	Highly recommended. Relatively low overhead. Care should be taken during data modeling in Ada.
Database	Highly recommended. Select SQL interface or other form of binding to database.

6.3 FUNCTION WRAPPING IN ADA

One of the simplest forms of wrapping is to call functions written in another programming language. In most cases, individual functions are compiled and mapped under the same operating environment using the same linker. The examples of this section illustrate how an Ada program calls a function or procedure that has been implemented in a different programming language. The structure of the various segment of the software is shown in Figure 19 on page 46.

Developing an interface to an external function callable from Ada requires the following information:

- The name of the routine
- The type of the call required
- The data type of each parameter
- The type of access required for each parameter

- The mechanisms needed to pass the parameters
- Whether any of the parameters are themselves routines or the addresses of the routines
- Whether or not any parameters are optional

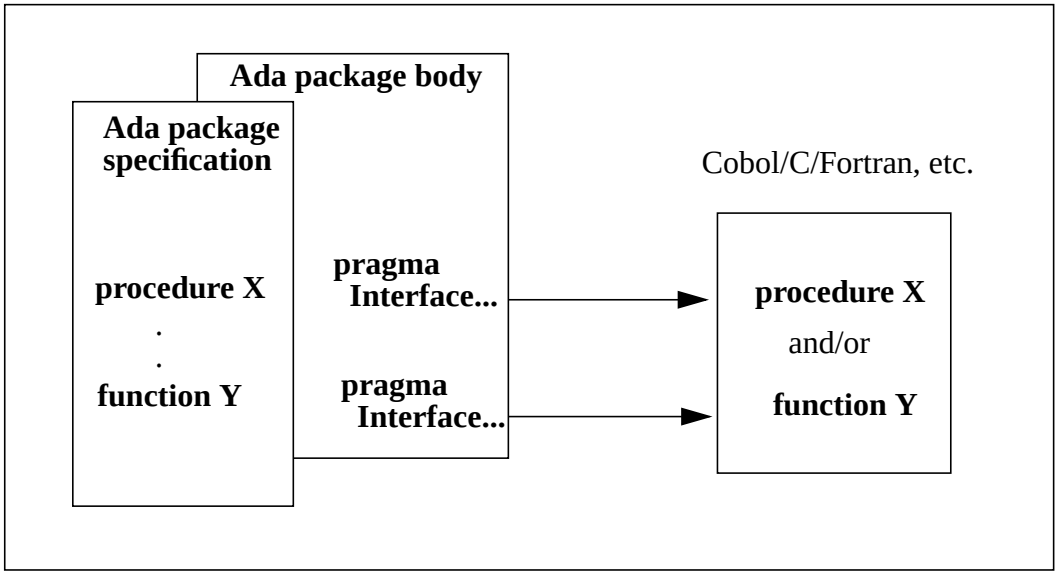

Figure 19. Calling a Wrapped Procedure/Function/Subprogram

Thus, one must transform the requirements in Ada terms, to create an equivalent Ada subprogram specification and use the **pragma INTERFACE** and any relevant Ada import pragmas to import the routine so that the programmer can call it as an Ada subprogram.

Ada supports two types of subprograms:

- Procedures, which can have parameters that are updated within the body of the subprogram.
- Functions, which return results but cannot update their parameters.

Wrapped routines must be imported into an Ada program before they can be used. In Ada 83, a generic pragma is provided to enable the programmer to import a routine developed in another programming language. The syntax of the pragma is

pragma INTERFACE (<Language_name>, <routine_name>)

The pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding routine. A body is not allowed for such a routine (not even in the form of a body stub) since the instructions of the routine are written in another language.

Vendors often extend this feature and provide additional pragmas to facilitate the bindings. For example, DEC Ada includes the following additional pragmas:

pragma IMPORT_PROCEDURE

pragma IMPORT_FUNCTION

When using this compiler, the pragma *INTERFACE* is used just to specify the name of the external routine, whether its a function or a procedure. Then one of these compiler-specific pragmas (*IMPORT_PROCEDURE*, or *IMPORT_FUNCTION*) is used to specify the details of the imported routine and its connection with a corresponding internal routine. More specifically, these pragmas include fields for both the internal and external routine names, the data types of the parameter values, and the mechanism for passing parameter values. The IMPORT_FUNCTION pragma also supports specification of the result data type. The examples of this section illustrate the use of these pragmas in wrapping external Cobol code.

Because many system and utility routines return results and update their parameters, DEC Ada provides an additional pragma specifically to import such subprograms. For example, in DEC Ada, the pragma *IMPORT_VALUED_PROCEDURE* in combination with the pragma *INTERFACE* enables the user to write an Ada interface that will import a routine so that it is interpreted as a procedure in the Ada environment and as a function in the external environment.

6.3.1 Example 1: “Employee_Taxable”

An example of function wrapping is shown in Figure 20 on page 48. In this example, an Ada procedure calls a function TAX_CALC written in Cobol. The calling procedure passes two parameters, *Emp_Income* and *Emp_Deduction*, to the called function. The called function then performs the calculation and returns the result. Integer data types are used for the purpose of simplicity. Cobol can support complex data types which can then be specified in Ada. Additional packages, such as ADAR (Ada Decimal Arithmetic and Representatives) developed by the Ada Joint Program Office, can be used to support data types not predefined in Ada.

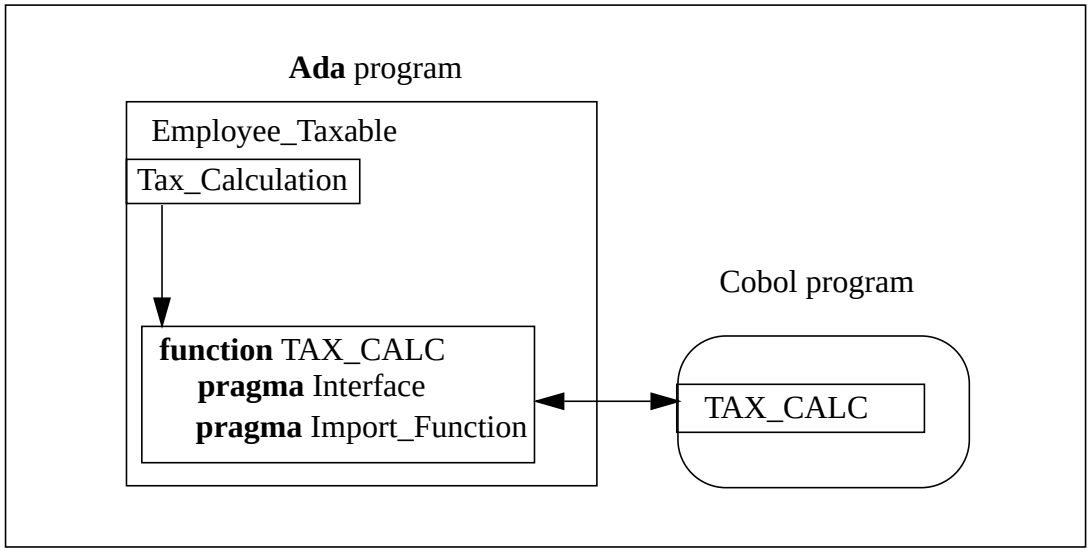

Figure 20. Function Wrapping Example in Ada.

The interface components of the calling Ada program are listed as follows:

```
with...;
package body Employee_Taxable is
    ...
    function TAX_CALC (EMP_INCOME, EMP_DEDUCTION: integer)
        return integer;
    pragma INTERFACE (Cobol, TAX_CALC);
    pragma IMPORT_FUNCTION (
        INTERNAL => TAX_CALC,
        EXTERNAL => TAX_CALC,
        RESULT_TYPE => INTEGER,
        PARAMETER_TYPE => (INTEGER, INTEGER),
        MECHANISM => (REFERENCE, REFERENCE));
    ...
    function Tax_Calculation (Self : in Class) return Employee.Money
        is
        Salary          : constant Employee.Money := Emp_Salary (Self);
        Emp_Income : constant Integer := Integer (Emp_Salary (Self));
        Emp_Deductions      : constant Integer := Deductions (Self);
    begin
        return Employee.Money (Tax_Calc (Emp_Income, Emp_Deduction));
```

```
    end Tax_Calculation;
    ...
end Employee_Taxable;
```

The called Cobol program is as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TAX_CALC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01          INCOME-TAX          PIC          9(9)          COMP.
01          TAX-RATE-1          PIC          9V99          VALUE     0.16.
01          TAX-RATE-2          PIC          9V99          VALUE     0.28.
01          TAX-RATE-1          PIC          9V99          VALUE     0.31.
LINKAGE SECTION.
01          EMP-INCOME          PIC          9(9)          COMP.
01          EMP-DEDUCTION       PIC          9(9)          COMP.
PROCEDURE DIVISION USING
            EMP-INCOME EMP-DEDUCTION GIVING INCOME-TAX.
BEGIN.
            IF EMP-INCOME <= 20000
                  COMPUTE INCOME-TAX =
                  (EMP-INCOME - EMP-DEDUCTION * 2500) * TAX-RATE-1
ELSE
            IF EMP-INCOME > 20000 AND <= 40000
                  COMPUTE INCOME-TAX =
                  (EMP-INCOME - EMP-DEDUCTION * 2500) * TAX-RATE-2
ELSE
                  COMPUTE INCOME-TAX =
                  (EMP-INCOME - EMP-DEDUCTION * 2500) * TAX-RATE-3.
EXIT PROGRAM.
```

6.3.2 Example 2: “Payroll”

This example is similar to Example-1 and here the type of the passed parameters is the character type.

Ada implementation for Payroll.Generate_Report routine is as follows:

```
package body Payroll is
```

```
        procedure REPORT_HEADER (TITLE: in out STRING);
        pragma Interface (Cobol, REPORT_HEADER);
        pragma Import_Procedure (
            INTERNAL => REPORT_HEADER,
            EXTERNAL => REPORT_HEADER,
            MECHANISM=> (Reference, Reference));
        -- Procedure to call Cobol routine
        procedure Generate_Report is
            Title: constant STRING(1.10) := "Division ";
        begin
            PAY_REPORT(Title);
        end Generate_Report;
        ...
    end Payroll;
```

The template for the wrapped Cobol program is as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT_HEADER.
DATA DIVISION.
LINKAGE SECTION.
01          REPORTTITLE          PIC          X(10).
PROCEDURE DIVISION USING REPORTTITLE.
BEGIN.
 DISPLAY "Report for " report_title.
```

The actual report generating code is not listed for reasons of simplicity.

```
 EXIT PROGRAM.
```

6.3.3 Example 3: "Math_Library"

In this example an Ada calling program call function is written in other native-mode languages. Simple integer addition is shown in the example, although any specialized math calculation could be wrapped similarly.

```
--Wrap a Math Library
package body Math_Library is
    function GETSUM (A, B: Integer) return Integer;
    pragma INTERFACE (FORTRAN, GETSUM);
```

```
    pragma IMPORT_FUNCTION (
        INTERNAL => GETSUM, -- Ada name
        EXTERNAL => GETSUM, -- external name
        RESULT_TYPE => INTEGER,
        PARAMETER_TYPES => (INTEGER, INTEGER),
        MECHANISM => (REFERENCE, REFERENCE));
    ...
    function "+"(A, B : Integer) return Integer is
    begin
        return GETSUM (A, B);
    end "+";
    ...
end Math_Library;
```

The Fortran-based called routine is as follows:

```
FORTRAN FUNCTION
integer function GETSUM (I, J)
C FUNCTIONAL DESCRIPTION:
C
C This FORTRAN function calculates the sum of 2 integers
C
 GETSUM = I + J
end
!
! This BASIC function calculates the sum of 2 integers
!
BASIC FUNCTION
function integer GETSUM (integer A,B)
 GETSUM = A + B
 functionend
;
; This MACRO function calculates the sum of 2 numbers
;
MACRO FUNCTION
.ENTRY GETSUM, ^M<> ; null entry mask
```

```
CLRQ R0 ; clear R0 and R1
ADDL3 @4(AP), @8(AP), R0 ; obtain sum
RET
.END
```

6.4 EXAMPLE SCENARIO

The main goal is to extend the life of an existing Cobol system that maintains company employee payroll information. This legacy system performs tax calculations, creates summary reports by departments and for the entire organization, and prints the reports. An additional goal is to facilitate the addition of new functions, such as printing employee checks. Figure 21 illustrates the architecture of this legacy system along with the planned OO system that will wrap it. The new system will generate the same reports and migrate to Ada. The first of several enhancements is to add the printing of payroll checks.

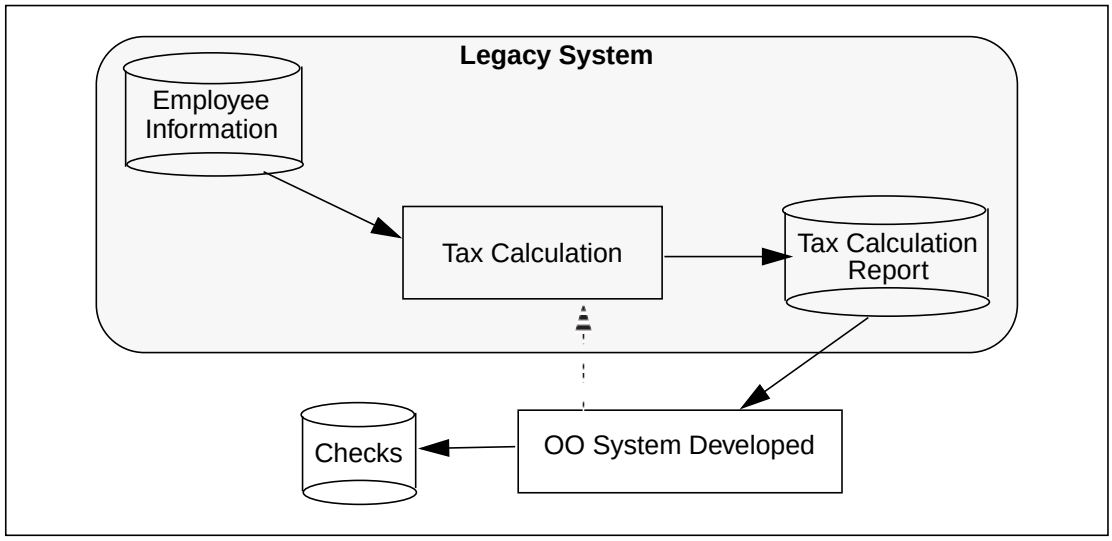

Figure 21. Wrapping of Scenario Legacy System

6.4.1 Legacy Program Scenario

The example legacy Cobol program finds data from the file TESTP.DAT and performs the tax calculation based on the status of the employee and number of deductions. It does not calculate any tax if an employee is a consultant. This input file contains ASCII text, as do the output files.The software generates a detailed report organized by department, including the name of the employee, the income, employment status, department number, number of dependents, and tax deducted from the employee's pay check,. It also prints out the total payroll and taxes for departments and for the company. The output of

the legacy system is written to the file PRINTER.DAT. The legacy program code is provided in Appendix A, Section A.1, along with the sample data file for input and output.

6.4.2 Migration Program Scenario

The new program will create the same detailed report, and print checks. If the employee is a consultant, the check is addressed to the home mailing address. Other employees receive their checks at work, and the checks are addressed to their departments. Check numbers are printed on the checks. The date printed on the checks is taken from the system date, and the name and net pay are provided by the legacy system using the existing program.

The migration program must be ready to accommodate a number of foreseeable changes. Some future enhancements may replace the input data file with an interface to a database. The company may begin using temporary employees provided through temp services. In addition, Accounting is looking into ways to authorize direct deposit transfers.

6.4.3 Object Model Scenario

Regardless of the language, the desire to migrate to an OO paradigm requires an object model. OO analysis of the relevant parts of the existing system and the potential enhancements produces an object model. The new architecture is represented via two models, namely, object model and dynamic model. Figure 22 on page 54 illustrates an object model diagram for the desired system.

The dynamic model that prints checks is illustrated by the object interaction diagram in Figure 23 on page 55. The other interaction is to generate the report. Depending on the OO methodology, these two interactions may be dependent on each other. For example, if the checks are printed based on the output of the report, then the report must obviously be printed first. However, if the checks are printed by reading the input file and invoking the Cobol tax calculation code to determine net pay, then no dependency exists.

6.4.4 Wrapping a Data File Scenario

Almost all legacy systems use file systems. Files are often used to exchange data from one part of the application to another part. Therefore, one approach to wrapping a legacy system is to pass data through the file system, and invoke the legacy code either from the operating system or through APIs.

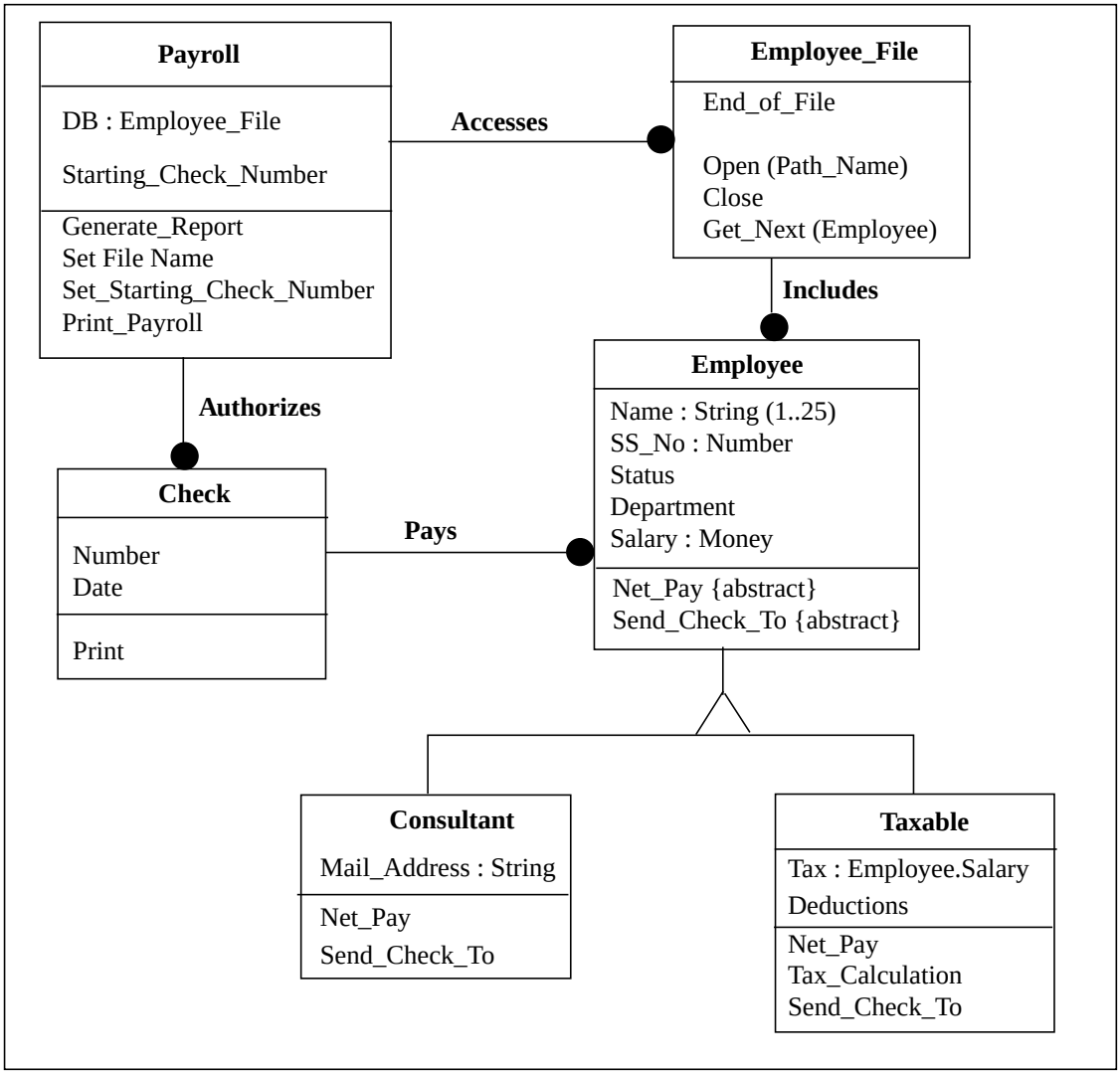

Figure 22. Object Model

The interaction between the legacy system originally developed in Cobol and how it is wrapped by an Ada 83 based OO system is shown in Figure 21 on page 52. The legacy system is invoked from the new system, and the report is used to populate the objects in the new system. The migration system is designed using OO concepts and implemented in Ada 83. The class-wide operation, Print_Checks, is used to illustrate run-time dispatching in Ada. The package specifications and package bodies are listed in Section A.2 of Appendix A.

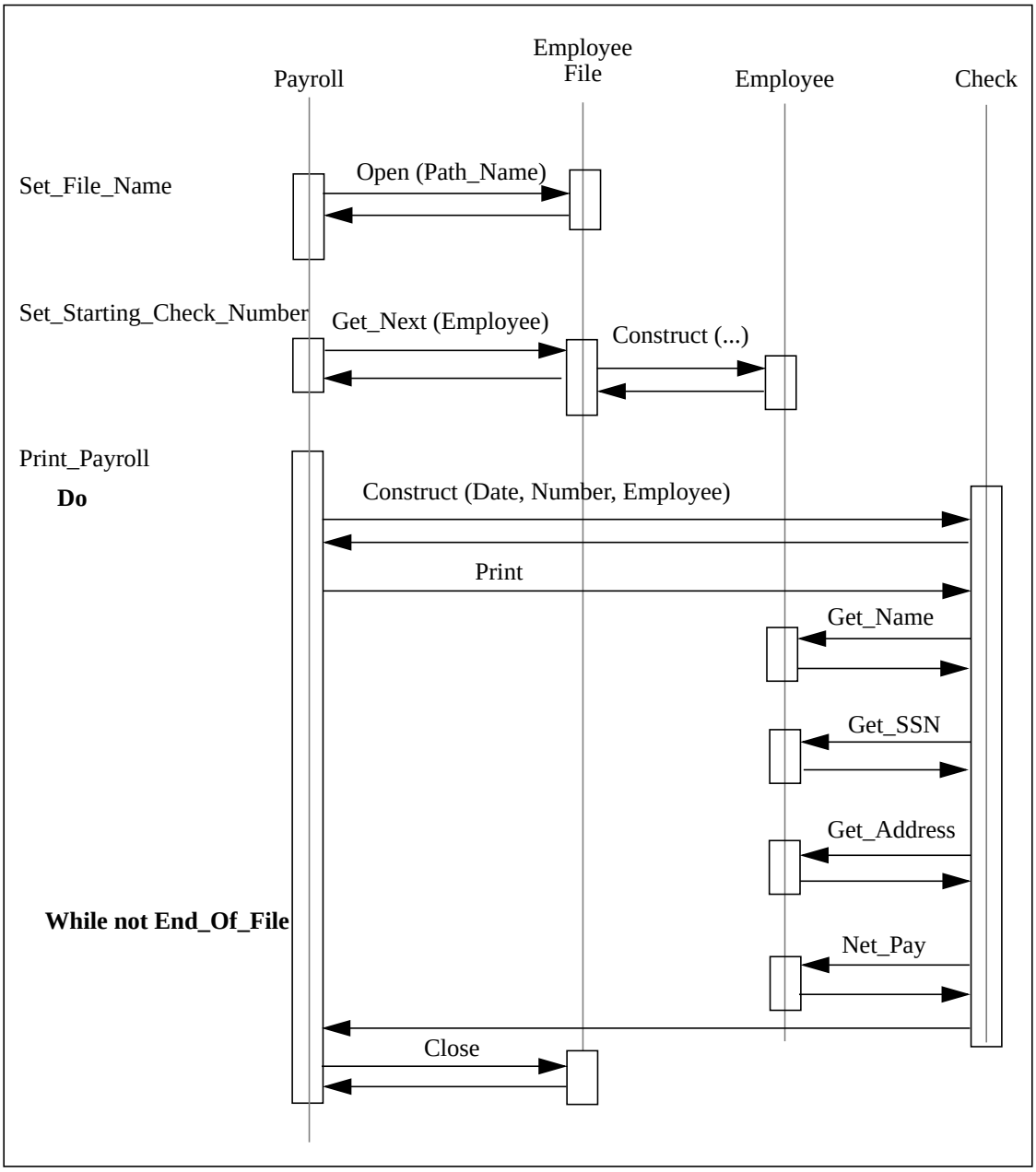

Figure 23. Interaction Diagram

6.5 INTERFACING TO EXTERNAL CODE

Support for interfaces to other languages using pragmas depends on the features of the Ada compiler and the implementation language of the "external code." The DEC Ada compiler used during the previous examples provides good interface support for many popular programming languages. Other compiler vendors may not provide such features in their environments. An alternative means of interfacing to "external code" may be required to achieve the same type of wrapping of legacy code within an OO system written in Ada. Three techniques for interfaces between Ada and foreign code are as follows:

- Interfacing via operating system
- Sharing common storage areas
- Interfacing through an intermediate programming language

When compiler pragmas are not available from Ada to a target language, then in most cases the compiler may support a set of APIs through which operating system services can be accessed. In such cases the operating system will invoke a separate software module implemented in another language.

When external function calls are not supported by the target Ada implementation, then sharing of information via common storage to exchange information between the called and the calling module can become a viable option. Programming languages like Fortran utilize common memory areas to exchange information among various software module during runtime. For time-critical applications, common memory can be utilized to transfer data between the called program developed in Ada to the calling legacy software developed in Fortran.

An intermediate programming language can provide another means to call foreign language code when an interface to the target code language is not directly supported by Ada compiler. In Unix systems, the C programming language often plays the role of an intermediate language since it is ordinarily supported within any Unix environment. Several examples are provided in the following subsections that implement the aforementioned techniques.

6.5.1 Operating System Interface

One way to establish a link from an Ada program to a block of legacy code written in another language, such as Cobol, is to interface through the underlying operating system. This form of wrapping is illustrated in Figure 24 on page 57. The figure shows an Ada pro-

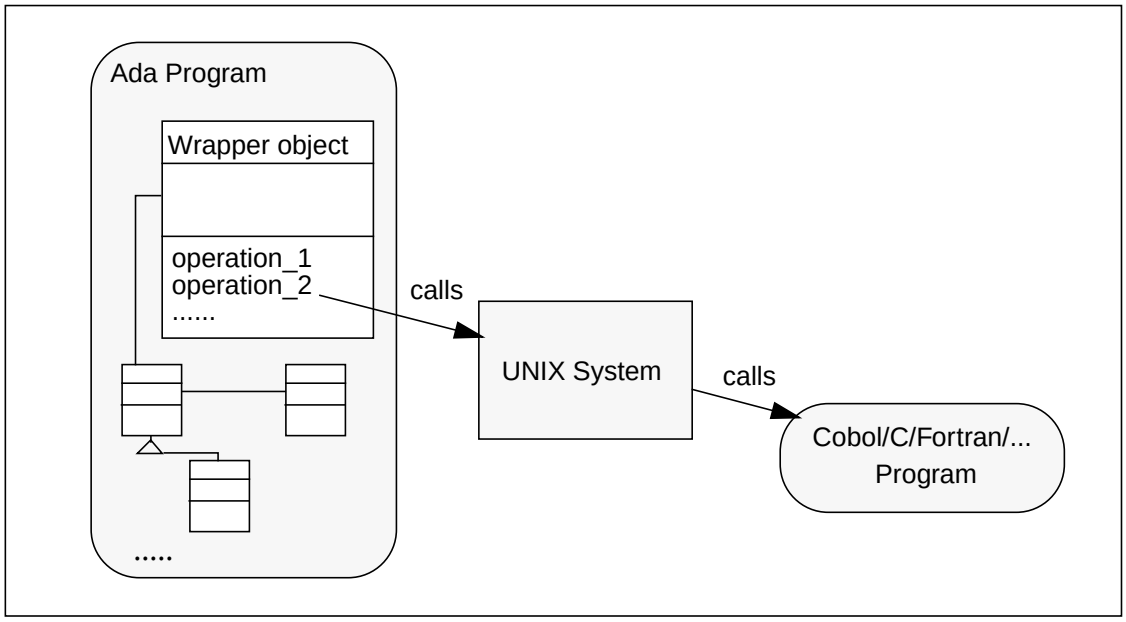

Figure 24. Linking via a Unix Shell

gram composed of an object wrapper and other related objects, with one of the wrapper object operations (*operation_2)* making a call to the Unix operating system which, in turn, executes an external program which could have been written in any supported programming language, such as Fortran, C, or Cobol.

Executing operating system (or shell) commands from within a program requires careful definition of the appropriate system calls to ensure format compatibility. Any such commands will naturally be environment dependent on such things as the compiler and operating system being used. For the purpose of illustration we define a package *Unix_Shell_Commands* that includes several basic commands for interacting with the operating system, all of which utilize externally defined C functions and data type definitions from the library package **System**. This package declares three basic operations, namely, *Unix_Shell*, *Get_Unix_Environment_Variable*, and *Get_Unix_Process_Id*.

```
package Unix_Shell_Commands is
    procedure Unix_Shell (cmd: in String; status: out Integer);
-- Execute the input string as a Unix shell command and return the
-- command's execution completion status.
-- A status value of zero usually
-- indicates successful command completion.
    function Get_Unix_Environment_Variable (Name: in String)
```

```
        return String;
    -- Retrieve the value of the named Unix environment variable.
    function Get_Unix_Process_Id return Integer;
    -- Retrieve the value of the program's process ID.
end Unix_Shell_Commands;
```

The body of the package then implements the operations beginning with a set of external C functions that are accessed via the *Interface* pragma, followed by the exported operations that utilize these external C functions:

```
with System; use System;
package body Unix_Shell_Commands is
    Unix_Null_Address: constant Address:= NO_Addr;
    --- Following are external C function that are imported: -- Retrieve the value of a Unix environment variable.
    -- Unix System library function.
    function getenv (Address_Of_Name_With_Null: in Address)
        return Address:
    pragma Interface (C, getenv);
    -- Retrieve the current program's Unix process ID.
    function Getpid return Integer;
    pragma Interface (C, getpid);
    -- Find the length of the multi-terminated string
    function strlen (Address_Of_String_With_Null: in Address)
    return Integer;
    pragma Interface (C, strlen);
    -- Execute a Unix Shell command
    function System (Address_Of_Cmd_With_Null: in Address)
        return Integer;
    pragma Interface (C, system);
    -- Utility to map the address of a string to a string result
    -- Retrieve the Ada string value of a Unix null-terminated string
    -- Note: Raises Storage_Error if address is null.
    function To_string (Address_Of_First_Char: in Address)
        return String is
        Result : String (1 .. strlen (Address_Of_First_Char));
        for Result use at Address_Of_First_Char;
```

```
    begin
        return Result;
    end To_string;
--- Following are the definitions for functions exported from this
--- package:
    function Get_Unix_Environment_variable (Name : in String)
        return String is
    -- Retrieve the value of the named Unix environment variable.
    Name_With_Null : constant String (1 .. Name'Length + 1) :=
        Name & ASCII.Null;
    Env_Var_Address : constant Address:=
        getenv (Name_With_Null(1)'Address);
    begin
        if Env_Var_Address = Unix_Null_Address then
            return "";
        else
            return To_String (Env_Var_Address);
        end if;
    end Get_Unix_Environment_Variable;

    function Get_Unix_Process_Id return Integer is
    begin
        return getpid;
    end Get_Unix_Process_Id;
    Procedure Unix_Shell (Cmd : in String; Status : out Integer) is
    -- Execute the input string as a Unix shell command and return the
    -- command's execution completion status. A status value of zero
    -- indicates successful command completion.
        Cmd_With_Null : constant String(1 .. Cmd'Length + 1):=
            Cmd & ASCII.Null;
    begin
        Status:= System(Cmd_With_Null(1)'Address);
    end Unix_Shell;
end Unix_Shell_Commands;
```

Note: The calling external programs in *Unix_Shell* take a string name of an external program as input and append a null character at the end before passing its address to the C function *System* which initiates execution of the code beginning at that address. It must also be noted that code interfacing a compiler to an operating system is ordinarily system specific, depending upon both the operating system and the compiler. This particular code was tested on only one combination of compiler and operating system. It illustrates the basic interface concept, although distinct code may be required for different compilers and operating systems.

6.5.2 Common Storage Areas Interface

When external code is called from Ada without the benefit of direct return values (as when executed via a Unix shell command), it may be necessary to provide some other means of communicating results from the external code to the Ada program. One obvious approach consists of simply writing the results from the external code to a file and reading those results back from the file after Ada resumes control. Writing and reading files, however, entails substantial overhead costs and may adversely affect the system's performance. An alternative is provided by some Ada compilers (e.g., DEC Ada) in the form of a pragma identifying certain data as defined in a common storage area accessible from other programs, as illustrated in Figure 25.

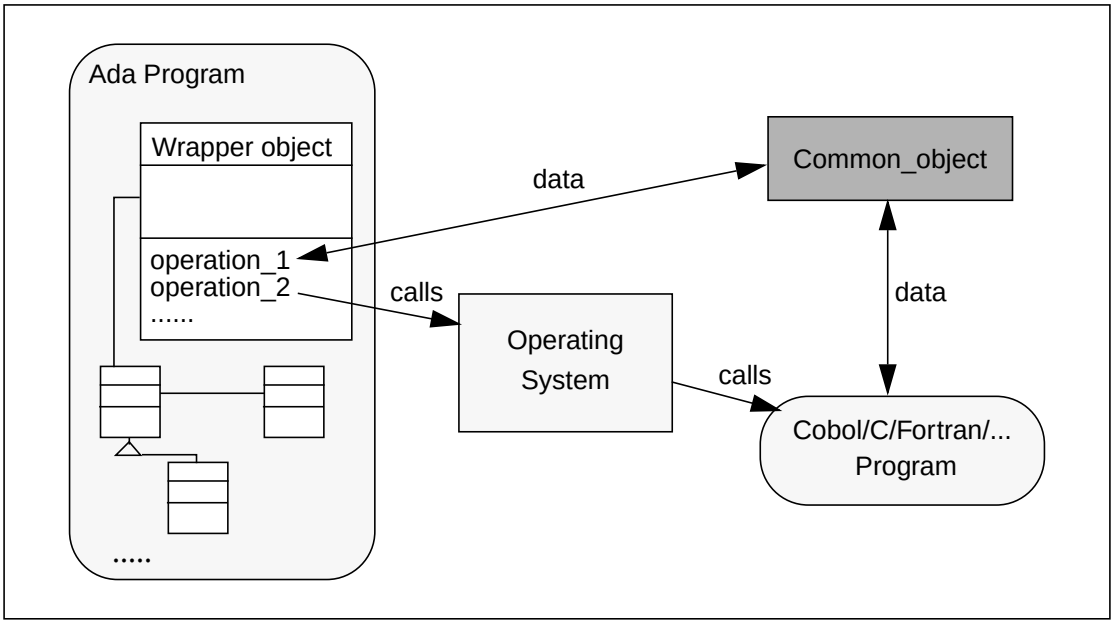

Figure 25. Interfacing Using a Common Area

In DEC Ada, the syntax for pragma for defining a common areas is as follows:

```
pragma COMMON_OBJECT (<internal_name> [, external_designator]
                      [, [SIZE =>] external_symbol]);
```

This pragma can be used to associate Ada storage with Fortran or Basic common blocks, Pascal variables declared with the COMMON or PSECT attribute, and EXTERNAL variables in PL/I or variables declared with the EXTERN declaration in C programs.

The following example illustrates how to share one storage area with several Fortran common variables with Ada record variables, where each field of the record corresponds to one Fortran variable.

```
C       FORTRAN declarations:
        INTEGER DAY, MONTH, YEAR
        CHARACTER*20 NAME
        COMMON //BDATE/DAY, MONTH, YEAR // NAME
        END
--
-- Corresponding Ada declarations;
package Birthdate_Interface is
    type DATE is
        record
            DAY, MONTH, YEAR: INTEGER;
        end record;
    subtype NAME is STRING (1 .. 20);
    procedure Next
        (Birthdate      : out Date;
        Account_Name    : out Name);
end Birthdate_Interface;

package body Birthdate_Interface is
    BDATE: DATE;
    ACCTNAME: NAME;
    pragma COMMON_OBJECT (BDATE);
    pragma COMMON_OBJECT (ACCTNAME, "$BLANK");

    procedure Next
```

```
        (Birthdate          : out Date;
         Account_Name       : out Name) is
    begin
        Birthdate := BDATE;
        Account_Name := ACCTNAME;
    end Next;
end Birthdate_Interface;
```

6.5.3 Intermediate Language Interface

A wrapped object can be called via an intermediate language that has a binding to Ada. Today, several vendors provide a binding to C. Thus, an Ada object can call a C function which in turn can call a Cobol/Fortran object. Such a wrapping strategy is illustrated in Figure 26.

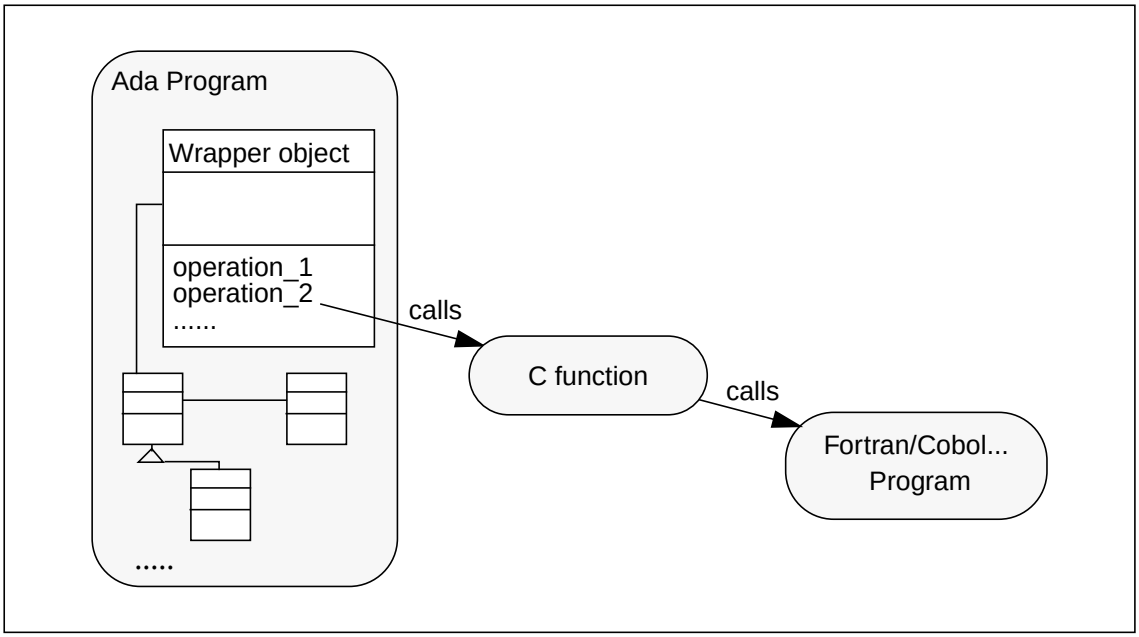

Figure 26. Interfacing via an Intermediate Language

6.5.3.1 Calling C from Ada

An example follows of wrapping a C function in Ada, which was implemented on the Suns using the Verdix Ada. A key thing to note is that Ada strings have their lengths managed separately, but C strings are terminated with '\0', so any program passing strings has to do a conversion. System Address is used in this example, and usually it is desirable to avoid doing that, but it was a simple way to implement this.

```
package Distance_Wrapper is
    function Distance (Left : String; Right : String) return Float;
end Distance_Wrapper;

with System;
package body Distance_Wrapper is
     -- Here's the declaration of the C "wrapped" function & types.
    function Miles_Distant(C1 : System.Address; C2 : System.Address)
        return Float;
    pragma Interface(C, Miles_Distant);
    -- Ada Language RM 13.9(4) allows limiting the types passable.
    -- Verdix requires parameter types to be scalar, access or
    -- SYSTEM.ADDRESS
    Miles_To_Kilometers : constant := 1.61;
    function Distance (Left : String; Right : String) return Float is
        Miles : Float;
        C1 : String (1..Left'Last+1);
        C2 : String(1..Right'Last+1);
    begin
        C1(1..Left'Last) := Left; C1(Left'Last+1) := ASCII.Nul;
        C2(1..Right'Last) := Right; C2(Right'Last+1) := ASCII.Nul;
        Miles := Miles_Distant(C1'Address, C2'Address);
        return Miles * Miles_To_Kilometers;
    end Distance;
end Distance_Wrapper;
```

6.5.3.2 Calling Fortran from C

Vendors generally place restrictions during interlanguage calls. For example, in VAX when one calls an external routine as a function, a single value is returned. When one calls an external routine as a subroutine (a VOID function), values are returned in the argument list. By default, VAX C passes all arguments by immediate value with the exception of arrays and functions; these are passed by reference.

The example that follows shows a VAX C function calling a VAX Fortran subprogram with a variety of data types. This example does not extend the previous one to illustrate the full concept of calling from Ada through C to Fortran, however, since an

environment supporting this full interface chain was not available to the authors at the time these examples were developed.

For most scalar types, VAX Fortran expects arguments to be passed by reference but character data is passed by descriptor:

```
/*
/* Beginning of C function
/*
#include <stdio.h>        /* get layout descriptors */
#include <descrip.h>      /* declare FORTRAN function */
extern int fort();
main()
{    int i = 508;
     float f = 649.0;
     double d = 91.5;
     struct {
               short s;
               float f;
            } s = {-2, -3.14};
auto $DESCRIPTOR (string1, “Hello, FORTRAN”);
struct dsc$descriptor_s string2;
/* “string1” is a FORTRAN-style string declare and initialized
/* using the $DESCRIPTOR macro.
/* string2 is also a FORTRAN-style string, but here we are
/* declaring and initializing by hand */
string2.dsc$b_dtype = DSC$K_DTYPE_T;   /* type is character */
string2.dsc$b_class = DSC$K_CLASS_S;   /* string descriptor */
string2.dsc$w_length = 3;   /* 3 characters */
string2.dsc$a_pointer = “bye”;   /* pointer to string value */
printf (“FORTRAN result is %d\n, fort (&i, &f, &d, &s, &string1,
                                        &string2));
}
/* end of C program */
C
C         Begin the FORTRAN subprogram
```

```
C
      INTEGER FUNCTION FORT (I, F, D, S, STRING1, STRING2)
      INTEGER I
      REAL F
      DOUBLE PRECISION  D
      STRUCTURE  /STRUCT/
      INTEGER*2 SORT
      REAL FLOAT
      END STRUCTURE
      RECORD  /STRUCT/ S
C     We can tell the program to use the length in the descriptor
C     or we can tell the program to ignore the descriptor and
C      assume the string has a particular length as done for
C      string2.
      CHARACTER*(*) STRING1
      CHARACTER*3 STRING2
      WRITE (5, 10) I, F, D, S.SORT, S.FLOAT, STRING1, STRING2
10    FORMAT (1X, I3, F8.1, D10.2, I7, F12.2, 1X, A, 2X, A)
      FORT = -15
      RETURN
      END
```

6.6 WRAPPING A DATABASE MANAGEMENT SYSTEM

Many legacy information management systems relied on databases to organize, protect, and store their data. Such database systems might have been designed by the in-house developers or might have been procured from commercial vendors. In either case, they were developed using the basic principles of DBMSs. Within DoD, a large number of legacy systems rely heavily on DBMSs to store and manage their data. Thus, the systems designers will have to provide a mechanism to encapsulate the legacy systems's DBMS. Today, most databases utilize some form of SQL to interface between an application and a DBMS. Although Ada 83 does not provide a direct binding to DBMS or SQL, one approach is to create a general binding following McCoy's strategy [MCC90]. The basic elements of the strategy include the following:

- Creation of Ada data type
- Interfacing to the external routines

- Interfacing to external data
- Linking to external library

Data types in Ada must be created to match those supported by the particular binding. This may require the use of an Ada representation clause to map a user type to one of the primitive types available in the SQL.

One can develop Ada subprogram specifications to match those of the interface language. A proper interface is then declared to map the Ada templates to the binding routines. This may be accomplished via pragma interfaces described earlier.

Interfacing to external data may involve the development of special routines that return the required data objects as parameters with proper format.

Linking to external library is implemented if a set of Cobol or C routines is available as a part of a library that provides a binding to DBMS. In such cases, proper linkage can be established by linking to available library routines.

6.6.1 SQL to Ada Binding

SQL has emerged as the industry standard for a relational data access language. SQL defines a common relational database language that enables consistency across product implementations, in the way users, application developers, and to some extent database designers interface with the products. Although internal mechanisms for representing and accessing database structures may vary greatly, SQL allows users to deal with one syntax for invoking those mechanisms.

The major problem with creating an SQL binding is that the developers of application software are now faced with two entirely different programming paradigms. A large amount of application software is procedural and functionally oriented, whereas the SQL part is used to model the relational algebra required for the database queries. SQL does not support the strong typing used in Ada; consequently, the application developers either use the limited set of types or must perform some form of a transformation.

The Ada community identified three viable options for implementing an Ada binding to SQL [SEI91, DON87]:

- All-Ada binding. SQL queries are modeled with standard Ada statements. For example, a relation expressed in SQL as a table will create an Ada record type.

In this case, SQL reserved words, such as **select** and **all** that are also Ada reserved words, are renamed in the Ada model.

- Embedded SQL. The SQL statements are included in the Ada application code to express the relations required for accessing the DBMS. A pre-processor is then used to translate the SQL statements to their equivalent Ada procedure calls that will actually make the queries to the DBMS.
- New language. To avoid the mixing of Ada and SQL, a new programming language SAMeDL (SQL Ada Module Description Language) has been proposed [SEI91]. The goal is to bridge the gap between Ada application oriented programming and SQL DBMS accesses. The SAME methodology requires that SQL statements be separated from the Ada application code and encapsulated in separate modules. The SQL statements are not embedded in the Ada packages, thus isolating the Ada application from the DBMS design and implementation. SAMeDL is designed to facilitate the construction of Ada database applications that use the SAME methodology.

These three techniques have their unique advantages and disadvantages, and the developer can only select the most appropriate approach suitable to the environment and application.

6.6.2 All-Ada Bindings

Today, many DBMS and Ada compiler vendors support the all-Ada bindings technique. The name of the Ada package specification is identical to the name of the SQL module. The procedures declared by the Ada specification have names identical to the corresponding procedures declared within the SQL module. The formal parameters of the procedures have names identical to those of the SQL module.

This technique is illustrated via an example which accesses a Parts-Supplier database [DATE75]. The simple SQL module contains a cursor declaration and the procedures open, fetch, and close. The Ada specification module contains the corresponding package specification. The Ada package Example_Definitions is a domain package in the terminology of [GRAH89], and represents a definitional module in the terminology of [CHAS90].

```
Module Example_Module
Language Ada
Authorization Public
```

```
Declare Part_City Cursor
For
    Select SP.PNO, S.City
    From SP, S
    Where SP.SNO = S.SNO
    And S.Status >= Input_Status;
Procedure Part_City_Open
    Input_Status Int
    SQLCODE;
    Open Part_City;
Procedure Part_City_Fetch
    Part_Number Char (5)
    City Char (15)
    City_Indic Smallint
    SQLCODE;
    Fetch Part_City into Part_Number, City INDICATOR City_Indic;
Procedure Part_City_Close
    SQLCODE;
    Close Part_City;
```

The specification of the Ada interface for the above SQL module is as follows:

```
with Example_Definitions;  use Example_Definitions;
package Example_Definitions is
    type Part_Nbr_City_pairs is
        record
            Part_Number      : Part_Number_Not_Null;
            City             : City_Type;
        end record;
    procedure Part_City_Open (Input_Status  : Status_Not_Null);
    -- creates the relation of part numbers and cities where there --- exists some
    -- supplier, with status at least Lower_Bound, of that part in --- that city.
    procedure Part_City_Fetch (
        Part_Cities   :   in   Part_Nbr_City_Pairs;
        Is_Found      :   out  Boolean);
    -- Returns the relation created by open
```

```
    -- Found becomes false at end of table
    procedure Part_City_Close;
    -- Clean up procedure
end Example_Interface;
```

The parameters to be passed between the application program and the SQL module should be carefully defined because in Ada the type equivalence is determined statically by name. The application program and the corresponding database management package must agree on the names as well as the structure.

6.6.3 Embedded SQL

Embedded SQL deals with the placement of SQL language constructs in procedural language code. Every vendor treats embedded SQL statements in a different way. For example, in Oracle's ProAda, software developers embed SQL statements directly in the Ada program and then precompile the source. Precompilation causes the embedded SQL to be translated into ProAda calls, including the runtime library procedures that handle the interaction between the application software and the Oracle relational DBMS.

For example, if the application wants to issue the following statement:

```
SELECT ename, sal
    FROM emp
    WHERE empno = &EMP_NUMBER
```

The equivalent embedded SQL statement in ProAda would be as follows:

```
EXEC SQL SELECT ename, sal
    INTO:EMPLOYEE_NAME,   :EMPLOYEE_SALARY
    FROM emp
    WHERE empno = :EMP_NUMBER;
```

In this case, the program must supply a valid employee number, placing it in the Ada host variable EMP_NUMBER, which must be declared and be in the scope of the embedded SQL statement. When the statement is executed, the name information and salary information that satisfy that query are placed into the Ada host variable EMPLOYEE_NAME and EMPLOYEE_SALARY.

The following example is a simple Ada program that connects to an Oracle DBMS; gets and prints the maximum employee number in the EMP table; selects and prints the department name for a user-provided department number, or prints an error if no such department number exists; and exits [ORCL92].

```
--    SIMPLE :

with text _io;
--    Note: the precompiler "with's"  the required ORACLE packages

procedure SIMPLE_SAMPLE is
     use text_io;

     --    declare host and program variables
     ORACLE_ID          : constant String := "SCOTT/TIGER";
     ENAME              : String (1..20);
     ENAME_LEN          : Integer;
     DEPT_NAME          :     String (1..14);
     LOCATION           :     String (1..13);

     SQL_ERROR          :     exception;
     SQL_WARNING        :     exception;

     --    Check to see if the last database
     --    operation returned any rows.

     function EMPLOYEE_EXITS return Boolean is
     begin
          return (not (ORACLE.ERROR.IF_NOT_FOUND));
      end EMPLOYEE_EXITS;

begin                              -- SIMPLE_SAMPLE
     --    Direct the precompiler to insert "if" logic that
     --    checks the ORACLE return code and raises an exception
     --    if needed.

     EXEC SQL WHENEVER SQLERROR raise SQL_ERROR;
     --    Check for warnings, such as data truncation, also.
     EXEC SQL WHENEVER SQLWARNING raise SQL_WARNING;

     --    Connect to ORACLE

     EXEC SQL CONNECT :ORACLE_ID;

     NEW_LINE;
     PUT_LINE ("Connected to ORACLE as " & ORACLE_ID);
     NEW_LINE;

     PUT_LINE ("*** ORACLE DEMO #1 ***");
     NEW_LINE;

     loop
          PUT ("Enter employee last name (CR to exit):";
          GET_LINE (ENAME, ENAME_LEN);
          exit when ENAME_LEN = 0;

          --    SELECT statements that return one row can use a
```

```
        -- simple SELECT statement. Otherwise, a cursor must be
        -- declared for the SELECT, and a FETCH statement is used.

        EXEC SQL SELECT INITCAP (loc), INITCAP (dname)
            INTO  :LOCATION,   :DEPT_NAME
            FROM emp, dept
            WHERE dept.deptno   =    empt.deptno
            AND EMP.ENAME =
                    (upper (:ENAME(1..ENAME_LEN));

        if EMPLOYEE_EXITS then
            NEW_LINE;
            PUT("Employee");
            PUT (ENAME(1..ENAME_LEN));
            PUT (" works for department " & DEPT_NAME);
            PUT (" in " & LOCATION);
            NEW_LINE;  NEW_LINE;
        else
            PUT_LINE (
                    "Sorry, no such employee (try ALLEN or JONES)");
            NEW_LINE;
        end if;
    end loop;

    NEW_LINE;
    PUT_LINE ("Bye-by.");

    -- Disconnect from the database.
    EXEC SQL COMMIT RELEASE;

exception
    -- Turn off error checking, since we do not want
    -- to raise an exception when logging out under
    -- any circumstance.

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL WHENEVER SQLWARNING CONTINUE;

    when SQL_ERROR =>
        PUT_LINE ("  ** ORACLE ERROR OCCURRED  **");
        NEW_LINE;
        PUT_LINE (ORACLE.ERROR.MESSAGE);
        EXEC SQL ROLLBACK RELEASE;
    when SQL_WARNING  =>
        PUT_LINE ("  ** ORACLE WARNING OCCURED  **");
        NEW_LINE;
        EXEC SQL ROLLBACK RELEASE;

end SIMPLE_SAMPLE;
```

Today, most of the database vendors such as Sybase, DEC, IBM, and Informix provide embedded SQL for their Ada compilers or offer such capabilities though their technology partners.

6.6.4 SQL Ada Module Description Language

The primary objective for SAMeDL is the partial creation of Ada DBMS applications where Ada applications are written without any mixed SQL statements, and SAMeDL modules are written to model the SQL queries. SAMeDL defines an abstract interface, a collection of Ada declarations through which an Ada program can access the DBMS. The meaning of a SAMeDL text is given by a translation into an Ada text, an SQL text, or both an Ada and an SQL text along with the relationship between them.

Figure 27 illustrates an overview of the SAMeDL architecture. A SAMeDL text

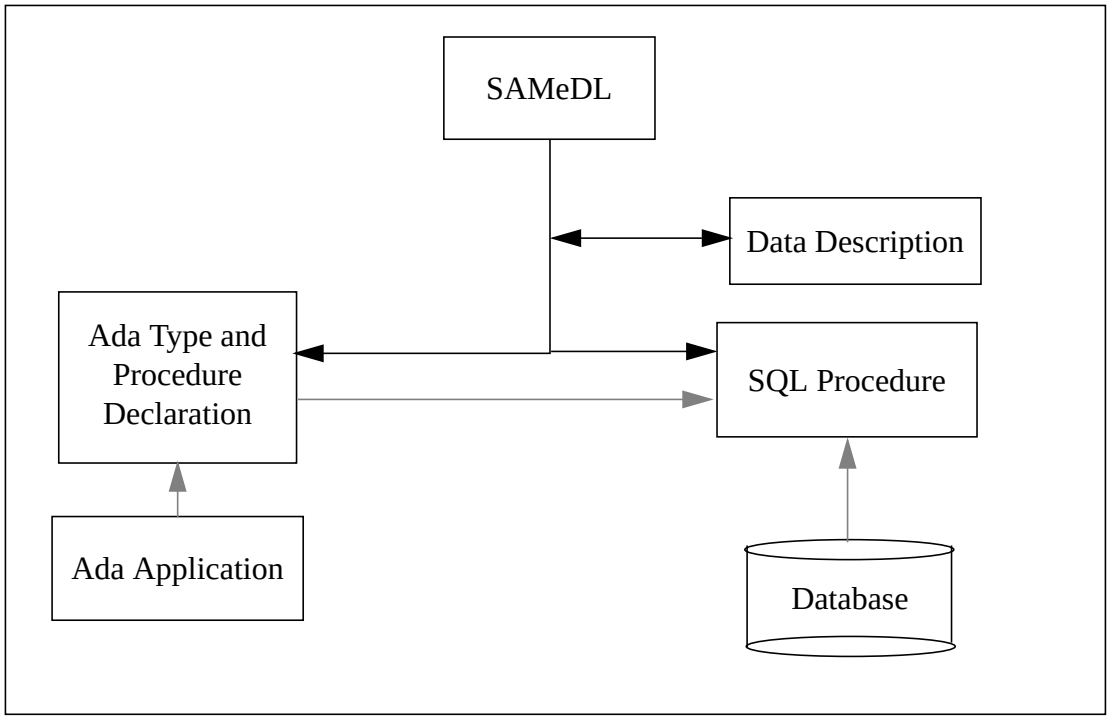

Figure 27. The Meaning of SAMeDL Text

may contain some data descriptions and it may also rely on previously processed data descriptions. The meaning of SAMeDL text may include Ada type and/or subprogram declarations. The actions of the subprograms nominally include calls to procedures defined in the SQL module language. The meaning of a SAMeDL procedure includes its Ada declaration, an SQL declaration, and the definitions of the input and output parameters of the procedures declared.

The following example illustrates the SAMeDL module for the SQL Module described in Section 6.6.2 on page 67.

```
with Example_Definitions;
abstract module Example_module is
authorization Public
    cursor Part_City
        (Input_Status: Status Not Null)
    for
        select SP.PNO Not Null named Part_Number, S.City
            from SP, S
            where SP.SNO = S.SNO
            and S.Status >= Input_Status;
    is
        procedure Part_City_Fetch is
            fetch into Part_Cities: new Part_Nbr_City_Pairs
            status Standard_Map;
    end Part_City;
```

The SAMeDL module does not generate an Ada package exactly. It generates an Ada package Example_Module containing a subpackage Part_City which, in turn, contains the declaration of a record type, Part_Nbr_City_pairs, and three procedures named Open, Fetch, and Close. The procedure Part_City_open in the SQL module has become Part_City.Open.

Recently, two software houses have announced commercial implementations of SAMeDL. Intermetrics in Cambridge, Massachusetts, has a SAMeDL version for Sybase. This version is going through beta testing and preliminary information looks very promising. It is likely that the binding will be sold and supported by the DBMS vendor (Sybase, Inc.), and the future implementation will be multi-threaded at the client site. Competence Center Informatik of Meppen, Germany, also has a version of SAMeDL for the Oracle database.

6.7 ADA 95 INTERFACE TO OTHER PROGRAMMING LANGUAGES

Ada 95 has eliminated the binding problem with other programming languages, most notably with Cobol and C. This section outlines the standard interface procedures for

Cobol taken directly from the Ada 95 Reference Manual [ANSI95]. In the interests of conserving space, not all of the interface operations provided for by the interface extension of Ada 95 are described here. The intent of this section is to outline enough of the new interface standard to provide the reader with clear expectations on its new interface capabilities.

6.7.1 Interfacing Pragmas

A pragma Import is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign language subprogram to be called from Ada, or a foreign language variable to be accessed from Ada. In contrast, a pragma Export is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. The *pragmas Import* and *Export* are intended primarily for objects and subprograms, although implementations are allowed to support other entities.

A pragma convention is used to specify that an Ada entity should use the conventions of another language. It is intended primarily for types and “callback” subprograms. For example, *pragma Convention (Fortran, Matrix);* implies that Matrix should be represented according to the conventions of the supported Fortran implementation, namely column-major order.

A *pragma Linker_Options* is used to specify the system linker parameters needed when a given compilation unit is included in a partition. An interfacing pragma is a representation pragma that is one of the pragmas Import, Export, or Convention. Their forms, together with that of the related *pragma Linker_Options*, are as follows:

```
pragma Import (
    [Convention      =>] convention_identifier,
    [Entity          =>] local_name [,
    [External_Name   =>] string_expression] [,
    [Link_Name       =>] string_expression]);

pragma Export (
    [Convention      =>] convention_identifier,
    [Entity          =>] local_name [,
    [External_Name   =>] string_expression] [,
    [Link_Name       =>] string_expression]);
```

```
pragma Convention(
    [Convention      =>] convention_identifier,
    [Entity          =>] local _name);

pragma Linker_Options(string_expression);
```

A pragma Linker_Options is allowed only at the place of a declarative_item. The expected type for a string_expression in an interfacing pragma or in pragma Linker_Options is String.

The example of interfacing to a pragma available in Ada 95 is as follows:

```
package Fortran_Library is
 function Sqrt (X : Float) return Float;
 function Exp (X : Float) return Float;
 private
 pragma Import(Fortran, Sqrt);
 pragma Import(Fortran, Exp);
 end Fortran_Library;
```

6.7.2 The Package “Interfaces”

The Ada 95 defined package called “Interfaces” is the parent of several library packages that declare types and other entities useful for interfacing to foreign languages. It also contains some implementation-defined types that are useful across more than one language (in particular for interfacing to assembly language).

The library package Interfaces has the following skeletal specification:

```
package Interfaces is
pragma Pure(Interfaces);
type Integer_n is range -2**n .. 2**n-1; --2’s complement
 type Unsigned_n is mod 2**n;
function Shift_Left (Value : Unsigned_n; Amount : Natural) return Unsigned_n;
 function Shift_Right (Value : Unsigned_n; Amount : Natural) return Unsigned_n;
 function Shift_Right_Arithmetic (Value : Unsigned_n; Amount : Natural)
     return Unsigned_n;
 function Rotate_Left (Value : Unsigned_n; Amount : Natural) return Unsigned_n;
 function Rotate_Right (Value : Unsigned_n; Amount : Natural) return Unsigned_n;
```

```
...
end Interfaces;
```

6.7.3 Interfacing with Cobol

6.7.3.1 Definitions

The facilities relevant to interfacing with the Cobol language are the package Interfaces.Cobol and support for the Import, Export, and Convention pragmas with convention_identifier Cobol.

The Cobol interface package supplies several sets of facilities:

- A set of types corresponding to the native Cobol types of the supported Cobol implementation (so-called “internal Cobol representations”), allowing Ada data to be passed as parameters to Cobol programs.
- A set of types and constants reflecting external data representations that might be found in files or databases, allowing Cobol-generated data to be read by an Ada program, and Ada-generated data to be read by Cobol programs.
- A generic package for converting between an Ada decimal type value and either an internal or external Cobol representation.

The library package Interfaces.Cobol is a child package of the package Interfaces, whose specification includes the following types of declarations:

```
package Interfaces.Cobol is
pragma Preelaborate(Cobol);
-- Types and operations for internal data representations
type Floating is digits implementation-defined;
type Long_Floating is digits implementation-defined;
type Binary is range implementation-defined;
...
function To_Cobol (Item : in String) return Alphanumeric;
function To_Ada (Item : in Alphanumeric) return String;
procedure To_Cobol (Item : in String;
                    Target : out Alphanumeric;
                    Last : out Natural);
procedure To_Ada (Item : in Alphanumeric;
                  Target : out String;
                  Last : out Natural);
```

```
...
-- Formats for Cobol data representations
type Display_Format is private;
Unsigned : constant Display_Format;
Leading_Separate : constant Display_Format;
Trailing_Separate : constant Display_Format;
...
-- Types for external representation of Cobol binary data
type Byte is mod 2**Cobol_Character'Size;
type Byte_Array is array (Positive range <>) of Byte;
pragma Pack (Byte_Array);
Conversion_Error : exception;
generic
type Num is delta <> digits <>;

package Decimal_Conversions is
-- Display Formats: data values are represented as Numeric
function Valid (Item : in Numeric;
                Format : in Display_Format) return Boolean;
function Length (Format : in Display_Format) return Natural;
function To_Decimal (Item : in Numeric;
                     Format : in Display_Format) return Num;
...
-- Binary Formats: external data values represented as Byte_Array
function Valid (Item : in Byte_Array;
                Format : in Binary_Format) return Boolean;
function Length (Format : in Binary_Format) return Natural;
...
-- Internal Binary formats: data values are of type Binary or Long_Binary
function To_Decimal (Item : in Binary) return Num;
function To_Decimal (Item : in Long_Binary) return Num;
...
end Decimal_Conversions;
private
 ... -- not specified by the language
```

```
end Interfaces.Cobol;
```

Each of the types in Interfaces.Cobol is Cobol compatible. The types Floating and Long_Floating correspond to the native types in Cobol for data items with computational usage implemented by floating point. The types Binary and Long_Binary correspond to the native types in Cobol for data items with binary usage, or with computational usage implemented by binary.

Each of the functions To_Cobol and To_Ada converts its parameter based on the mappings Ada_To_Cobol and Cobol_To_Ada, respectively. The length of the result for each is the length of the parameter, and the lower bound of the result is 1. Each component of the result is obtained by applying the relevant mapping to the corresponding component of the parameter.

Each of the procedures To_Cobol and To_Ada copies converted elements from Item to Target, using the appropriate mapping (Ada_To_Cobol or Cobol_To_Ada, respectively). The index in *Target* of the last element assigned is returned in *Last* (0 if Item is a null array). If *Item'Length* exceeds *Target'Length*, then *Constraint_Error* is propagated.

6.7.3.2 Example

One of the examples of calling a Cobol program from Ada 95 provided in the Ada 95 Reference Manual is as follows:

```
with Interfaces.Cobol;
 procedure Test_Call is
-- Calling a foreign Cobol program
 -- Assume that a Cobol program PROG has the following declaration
 -- in its LINKAGE section:
 -- 01 Parameter-Area
 -- 05 NAME PIC X(20).
 -- 05 SSN PIC X(9).
 -- 05 SALARY PIC 99999V99 USAGE COMP.
 -- The effect of PROG is to update SALARY based on some algorithm

package Cobol renames Interfaces.Cobol;
type Salary_Type is delta 0.01 digits 7;
type Cobol_Record is record
Name : Cobol.Numeric(1..20);
```

```
SSN : Cobol.Numeric(1..9);
Salary : Cobol.Binary; -- Assume Binary = 32 bits
end record;
pragma Convention (Cobol, Cobol_Record);
procedure Prog (Item : in out Cobol_Record);
pragma Import (Cobol, Prog, "PROG");
package Salary_Conversions is
new Cobol.Decimal_Conversions(Salary_Type);
Some_Salary : Salary_Type := 12_345.67;
Some_Record : Cobol_Record :=
(Name => "Johnson, John ",
 SSN => "111223333",
 Salary => Salary_Conversions.To_Binary(Some_Salary));
begin
 Prog (Some_Record);
 ...
end Test_Call;
```

This example could easily be modified to illustrate wrapping with a domain object. The Ada procedure *Prog* could be defined within a package that specifies a suitable object class, such as an employee class, and take such employee objects as an argument. The Ada *Prog* would need some modification to extract the relevant fields from the employee object and place them in the proper format of a record for the Cobol *PROG*. Then, the "legacy" Cobol procedure *PROG* would be wrapped by the employee class and the interface pragmas. We hesitate to present the actual code for such a modification since we have not been able to test it on an Ada 95 compiler yet, and we are limiting our listings of code fragments to those that have been tested or officially sanctioned (as in this last listing).

7. SUMMARY OF GUIDELINES AND ISSUES

7.1 GUIDELINES FOR OO WRAPPING

In the course of investigating alternative strategies and tactics for OO wrapping, a number of guidelines have emerged for choosing and applying them in a variety of contexts. OO wrapping has been identified as an effective technique for encapsulating legacy software components within a partially modernized migration system. Wrapping can support staged migration of legacy systems to modernized OO systems as well as the incorporation of trusted legacy software into new systems. Another application for which wrapping is recommended is to establish data standardization of legacy code and data without reengineering legacy systems.

When the resources are available, it is recommended to use domain object models for wrapping legacy components rather than simply wrapping components as software objects. Such object model wrapping is identified as providing a better foundation for any subsequent legacy modernization or extensions. Costs of building such object models can be minimized by judicious abstraction of the domain objects, modeling only those features that are essential to wrapping.

One reason behind favoring wrapping over reengineering is the presence of any strong time pressure to modernize a legacy system quickly; factors include the following:

- Expiring hardware and software contracts
- Shift to new platforms
- New functionality requirements
- Requirements for interoperability with other reengineered AISs
- Data item standardization requirements

Other reasons to prefer wrapping to reengineering are as follows:

- Absence of documentation
- Departure of all domain experts

- Complexity of code
- Fragility (or brittleness) of code
- Size of code or database
- Staffing resource limitations

Wrapping feasibility depends on conditions of the legacy and target migration environments, such as modularity of legacy code, and support for interfaces between legacy components and the migration OO environment. Under such favorable conditions, wrapping may be the most effective means of meeting modernization deadlines.

Guidance on overall system migration strategies is also provided. Four different such strategies are identified. The "one-shot rebuild" strategy is identified as risky for large systems because it attempts too much reengineering at once. Of the remaining strategies, "unite-and-conquer" stands out as generally superior due to its use of a unifying object model of the business enterprise for wrapping multiple software components. These models can provide transparent access to the data stores throughout the whole migration process. This supports incremental modernization of the legacy system while minimizing costly revisions to object models and data access code. Business models also provide a new OO perspective on the business domain that can be helpful in guiding subsequent modernization phases. Business model objects may also experience considerable reuse at subsequent phases of migration, and possibly even in other systems, thus lowering costs of subsequent migration activities. The only drawback to this strategy is the cost of building the business model. Hence, this strategy is only recommended for migration stages wherein sufficient resources are available for this extensive task. In other contexts, a less cohesive modernization of a legacy system may be all that is feasible.

7.2 LEGACY WRAPPING ISSUES

Wrapping legacy software offers considerable promise of easing the difficult transition from obsolete legacy systems to modernized systems with the advantages of greater maintainability, modifiability, and reuse inherent in OO technology. Our investigations into alternative strategies for designing and implementing software wrapping have identified a variety of issues which will benefit from further investigation.

Client-server model. This is a powerful paradigm for integrating legacy system components with modernized ones. Guidelines need to be developed on conditions under which the client-server should be considered and how it should be implemented. The guide-

lines should include the issue of migration from a mainframe-based legacy system to local area network based computing with open server, workstation, and communication protocols.

GUI-based front end. The GUI has become the standard interface for today's user. The issue of transforming the character-based user interface inherent to many legacy systems to the GUI must be addressed. In many legacy systems, multiple types of user interfaces exist based on the terminal types. Techniques need to be developed and guidelines must be prepared to map these terminal-oriented user interfaces to a single GUI.

Multiple terminals. Many legacy applications are tightly coupled to terminal hardware and proprietary communications software. Guidelines must be developed to help implementors create virtual terminals based on open communications protocols and software so that they can be directly mapped to today's GUI but still retain the look and feel of the user interface of the legacy system when warranted.

Database bindings. OO program bindings to relational DBMSs are still evolving and developers will need more standards-based technology support in this area. Technology must be developed so that one can capture the database of the legacy system and transition it to the new relational DBMS technology. The developers will also require guidelines and examples on how to interface their specific relational DBMSs to Ada programs which may or may not be supported by the vendors.

Data structure and conversion. Ada's arithmetic facility does not readily handle the exact decimal model needed for financial computations, leaving an impediment to a successful transition to Ada for information systems applications. The Ada Decimal Arithmetic and Representatives (ADAR) project addresses this shortcoming by providing a set of packages that define and implement decimal support in Ada 83. Today, most of the vendors do not support ADAR packages, and guidelines must be developed on the implementation and usage of financial data in Ada.

Interoperability. Issues related to interoperability between old, conventional systems and new, object-oriented systems need to be addressed for DoD. Is the Object Management Group's Object Request Broker, for example, a viable solution to the problem of interoperability?

Real-time legacy system. The use of OO technology for real-time system is not addressed in this report nor is the wrapping strategy for such systems. Further work needs

to be done that identifies the unique issues related to real-time system and how to resolve some of them. This is still an open research area.

APPENDIX A.
EXAMPLES OF OO PROGRAMMING CODE

A.1 LEGACY COBOL PROGRAM

A.1.1 COBOL Listing

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TAX-CALCULATION.
AUTHOR. UNKNOWN.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX.
OBJECT-COMPUTER. VAX.
INPUT-OUTPUT SECTION.

FILE-CONTROL.
        SELECT EMPLOYEE-FILE ASSIGN TO “TESTP.DAT”.
        SELECT DEPT-DIVISION-LIST ASSIGN TO PRINTER.
DATA DIVISION.

FILE SECTION.
FD      EMPLOYEE-FILE
        LABEL RECORDS ARE STANDARD
        DATA RECORD IS INPUT-RECORD.
01      INPUT-RECORD.
            02    IR-NAME          PIC    X(25).
            02    IR-ADDRESS       PIC    X(30).
            02    IR-NUMBER        PIC    9(9).
            02    IR-DIVISION      PIC    X.
            02    IR-DEPARTMENT    PIC    X(2).
            02    IR-SALARY        PIC    9(6).
            02    FILLER           PIC    X(2).
```

```
           02    IR-DEPENDENT                PIC          99.
           02    FILLER          PIC         X.
           02    IR-STATUS       PIC         X.
           02    FILLER          PIC         X.

FD         DEPT-DIVISION-LIST
           LABEL RECORDS ARE OMITTED
           DATA RECORD IS PRINT-LINE.
01      PRINT-LINE                 PIC      X(132).

WORKING-STORAGE SECTION.
01 CONSTANTS.
        02    LINES-PER-PAGE              PIC   99 VALUE 46.
01 COUNTERS.
        03    PAGE-CNT                    PIC   999 VALUE ZERO.
        03    LINE-CNT                    PIC   99 VALUE 46.
        88    PAGE-FULL                   VALUE  46 THROUGH 99.

01      ACCUMUL-TOTALS.
        03    NUMBER-OF-EMPLOYEES         PIC   9(3) VALUE ZERO.
        03    SALARY-TOTAL                PIC   9(8) VALUE ZERO.
        03    DEPT-NUMBER-OF-EMP          PIC   99 VALUE ZERO.
        03    DEPT-TOTAL-SALARY           PIC   9(8) VALUE ZERO.
        03    DEPT-TOTAL-TAX              PIC   9(8) VALUE ZERO.
        03    DIVISION-NUMBER-OF-EMP      PIC   9(4) VALUE ZERO.
        03    DIVISION-TOTAL-SALARY       PIC   9(8) VALUE ZERO.

01      HOLD-FILEDS.
        03    HOLD-DEPARTMENT             PIC   X(2) VALUE SPACES.
        03    HOLD-DIVISION               PIC   X VALUE SPACES.
        03    HOLD-STATUS                 PIC   X VALUE SPACES.

01      TAX-FIELDS.
        03    HOLD-FED-TAX                PIC   9(5)V99 VALUE ZERO.
        03    HOLD-STATE-TAX              PIC   9(5)V99 VALUE ZERO.
        03    TOTAL-FED-TAX               PIC   9(8)V99 VALUE ZERO.
        03    HOLD-TOTAL-TAX              PIC   9(8)V99 VALUE ZERO.
```

```
           03   TOTAL-STATE-TAX              PIC  9(8)V99 VALUE ZERO.
           03   HOLD-DEPENDENT               PIC  99       VALUE ZERO.
           03   NET-SALARY                   PIC  9(8)V99 VALUE ZERO.
           03   FED-TAX-RATE-1               PIC  9V99     VALUE 0.15.
           03   FED-TAX-RATE-2               PIC  9V99     VALUE 0.28.
           03   FED-TAX-RATE-3               PIC  9V99     VALUE 0.31.
           03   STATE-TAX-RATE               PIC  9V99     VALUE 0.05.

01         END-OF-FLAG                       PIC  X(3) VALUE "NO".
           88   END-OF-FILE                    VALUE "YES".

01         MAJOR-HEADING.
02         FILLER                  PIC  X(44)  VALUE  SPACES.
           02   FILLER             PIC  X(3)   VALUE   "ABC".
           02   FILLER             PIC  X      VALUE  SPACES.
           02   FILLER             PIC  X(7)   VALUE  "COMPANY".
           02   FILLER             PIC  X      VALUE  SPACES.
           02   FILLER             PIC  X(19)   VALUE
                                               "DIVISION/DEPARTMENT".
           02   FILLER             PIC  X      VALUE  SPACES.
           02   FILLER             PIC  X(8)   VALUE  "EMPLOYEE".
           02   FILLER             PIC  X      VALUE  SPACES.
           02   FILLER             PIC  X(6)   VALUE   "REPORT".
           02   FILLER             PIC  X(10)  VALUE  SPACES.
           02   FILLER             PIC  X(5)   VALUE  "PAGE ".
           02   MH-PAGE-COUNTER         PIC  ZZZ.

01         SUBHEADING.
           02   FILLER             PIC  X(25)  VALUE  " EMPLOYEE NAME ".
           02   FILLER             PIC  X(8)   VALUE  SPACES.
           02   FILLER             PIC  X(15)  VALUE  "EMP NUMBER".
           02   FILLER             PIC  X(2)   VALUE  SPACES.
           02   FILLER             PIC  X(6)   VALUE  "STATUS".
           02   FILLER             PIC  X(2)   VALUE  SPACES.
           02   FILLER             PIC  X(10)  VALUE  "DEPARTMENT".
           02   FILLER             PIC  X(3)   VALUE  SPACES.
           02   FILLER             PIC  X(5)   VALUE  " DEP ".
```

```
           02   FILLER              PIC   X(8)      VALUE     SPACES.
           02   FILLER              PIC   X(11)     VALUE     " SALARY ".
           02   FILLER              PIC   X(8)      VALUE     SPACES.
           02   FILLER              PIC   X(7)      VALUE     "TOT-TAX".

01         DETAIL-LINE.
           02   DL-NAME             PIC   X(25).
           02   FILLER              PIC   X(8)      VALUE     SPACES.
02         DL-EMPLOYEE-NUMBER       PIC   X(9).
           02   FILLER              PIC   X(9)      VALUE     SPACES.
           02   DL-STATUS           PIC   X.
           02   FILLER              PIC   X(7)      VALUE     SPACES.
           02   FILLER              PIC   X(6)      VALUE     SPACES.
           02   DL-DEPARTMENT       PIC   X(2).
           02   FILLER              PIC   X(5)      VALUE     SPACES.
           02   DL-DEPENDENT        PIC   Z9.
           02   FILLER              PIC   X(12)     VALUE     SPACES.
           02   DL-SALARY           PIC   $ZZZ,ZZZ.99.
           02   FILLER              PIC   X(8)      VALUE     SPACES.
           02   DL-TAX              PIC   $ZZZ,ZZZ.99.

01         DEPARTMENT-TOTAL-LINE.
           02   FILLER              PIC   X(35)     VALUE     SPACES.
           02   DTL-NO-EMP          PIC   Z9.
           02   FILLER              PIC   X(5)      VALUE     SPACES.
           02   FILLER              PIC   X(9)      VALUE     "EMPLOYEES".
           02   FILLER              PIC   X(27)     VALUE     SPACES.
           02   DTL-TOTAL-SALARY    PIC   $**,***,***.99.
           02   FILLER              PIC   X(8)      VALUE     SPACES.
           02   DTL-TAX             PIC   $***,***.99.

01         DIVISION-TOTAL-LINE.
           02   FILLER              PIC   X(35)     VALUE
                                                   " ****** DIVISION TOTAL ******".
           02   FILLER              PIC   X(10)     VALUE     SPACES.
           02   DVT-NUMBER-OF-EMP    PIC   Z9.
           02   FILLER              PIC   X(5)      VALUE     SPACES.
```

```
           02   FILLER                 PIC   X(9)    VALUE    "EMPLOYEES".
           02   FILLER                 PIC   X(7)    VALUE    SPACES.
           02   DVT-TOTAL-SALARY       PIC   $**,***,***.

01         SUMMARY-LINE.
           02   FILLER                 PIC   X(15)   VALUE    SPACES.
           02   FILLER                 PIC   X(20)   VALUE
                                                    "COMPANY TOTAL *****".
           02   SL-NO-OF-EMPLOYEES     PIC   ZZZ9.
           02   FILLER                 PIC   X(5)    VALUE    SPACES.
           02   FILLER                 PIC   X(9)    VALUE    "EMPLOYEES".
           02   FILLER                 PIC   X(37)   VALUE    SPACES.
           02   SL-SALARY-TOTAL        PIC   $**,***,***.99.

PROCEDURE DIVISION.

001-START-SECTION.
           PERFORM A200-INITIALIZATION.
           PERFORM      A300-CONTROL
           UNTIL END-OF-FILE.
           PERFORM A900-TERMINATION.
           STOP RUN.

A200-INITIALIZATION.
           OPEN INPUT EMPLOYEE-FILE
           OUTPUT DEPT-DIVISION-LIST.
           PERFORM     A400-READ.
                       MOVE IR-DEPARTMENT TO HOLD-DEPARTMENT
           MOVE IR-DIVISION TO HOLD-DIVISION.

A300-CONTROL.
           IF IR-DIVISION NOT = HOLD-DIVISION
           PERFORM A600-DEPARTMENT-BREAK
           PERFORM A650-DIVISION-BREAK
           ELSE
           IF              IR-DEPARTMENT NOT = HOLD-DEPARTMENT
                                               PERFORM A600-DEPARTMENT-BREAK.
```

```
        PERFORM A500-PROCESS.
        PERFORM A400-READ.

A400-READ.
        READ EMPLOYEE-FILE
        AT END MOVE “YES” TO END-OF-FLAG.

A500-PROCESS.

        MOVEIR-NAME                     TO    DL-NAME.
        MOVE        IR-NUMBER           TO    DL-EMPLOYEE-NUMBER.
        MOVE         IR-SALARY          TO    DL-SALARY.
        MOVE        IR-STATUS           TO    DL-STATUS.
        MOVE        IR-DEPENDENT        TO    DL-DEPENDENT.
        MOVE        IR-DEPARTMENT       TO    DL-DEPARTMENT.
        MOVE        IR-STATUS           TO    HOLD-STATUS.
        MOVE        IR-DEPENDENT        TO    HOLD-DEPENDENT.
        ADD         IR-SALARY           TO    DEPT-TOTAL-SALARY.
        ADD         IR-SALARY           TO    DIVISION-TOTAL-SALARY.

        IF          HOLD-STATUS NOT = “C”
        THEN
        PERFORM A550-TAX-CALCULATION
        MOVE        HOLD-TOTAL-TAX      TO    DL-TAX

        ADD         HOLD-TOTAL-TAX      TO    DEPT-TOTAL-TAX
            ELSE
                     MOVE ZERO TO DL-TAX.

        ADD         1                   TO    DEPT-NUMBER-OF-EMP.
        ADD         1                   TO    DIVISION-NUMBER-OF-EMP.
        ADD         IR-SALARY           TO    SALARY-TOTAL.
        ADD         1                   TO    NUMBER-OF-EMPLOYEES.

                    IF                  PAGE-FULL
                                        PERFORM A700-WRITE-HEADINGS.
                    WRITE PRINT-LINE FROM DETAIL-LINE AFTER 2.
```

```
        ADD 2 TO LINE-CNT.

A550-TAX-CALCULATION.
        IF      IR-SALARY <= 20000
         THEN
                COMPUTE NET-SALARY = IR-SALARY - (2500 * HOLD-DEPENDENT)
                COMPUTE HOLD-FED-TAX = FED-TAX-RATE-1 * NET-SALARY
        ELSE
        IF IR-SALARY > 20000 AND <= 40000
                COMPUTE NET-SALARY = IR-SALARY - (2500 * HOLD-DEPENDENT)
                COMPUTE HOLD-FED-TAX = FED-TAX-RATE-2 * NET-SALARY
        ELSE
                COMPUTE NET-SALARY = IR-SALARY - 2500 * HOLD-DEPENDENT
                COMPUTE HOLD-FED-TAX = FED-TAX-RATE-3 * NET-SALARY.

                COMPUTE HOLD-STATE-TAX = STATE-TAX-RATE * IR-SALARY.
                COMPUTE HOLD-TOTAL-TAX = HOLD-FED-TAX + HOLD-STATE-TAX.

A600-DEPARTMENT-BREAK.
            MOVE DEPT-NUMBER-OF-EMP TO DTL-NO-EMP.
            MOVE DEPT-TOTAL-SALARY TO DTL-TOTAL-SALARY.
            MOVE HOLD-TOTAL-TAX TO DTL-TAX.
            IF PAGE-FULL
                    PERFORM A700-WRITE-HEADINGS.

                WRITE PRINT-LINE FROM DEPARTMENT-TOTAL-LINE AFTER 2.
        ADD 2 TO LINE-CNT.
        MOVE 0 TO DEPT-NUMBER-OF-EMP.
        MOVE 0 TO DEPT-TOTAL-SALARY.
                MOVE IR-DEPARTMENT TO HOLD-DEPARTMENT.

A650-DIVISION-BREAK.
                MOVE DIVISION-TOTAL-SALARY TO DVT-TOTAL-SALARY.
                MOVE DIVISION-NUMBER-OF-EMP TO DVT-NUMBER-OF-EMP.
        IF PAGE-FULL
        PERFORM A700-WRITE-HEADINGS.
                WRITE PRINT-LINE FROM DIVISION-TOTAL-LINE AFTER 3.
```

```
            MOVE 0 TO DVT-NUMBER-OF-EMP.
            MOVE 0 TO DIVISION-TOTAL-SALARY.
            MOVE IR-DIVISION TO HOLD-DIVISION.

        A700-WRITE-HEADINGS.
            ADD 1 TO PAGE-CNT.
            MOVE PAGE-CNT TO MH-PAGE-COUNTER.
            WRITE PRINT-LINE FROM SUBHEADING AFTER 2.
            MOVE 3 TO LINE-CNT.

        A800-WRITE-SUMMARY-LINE.
                MOVE NUMBER-OF-EMPLOYEES TO SL-NO-OF-EMPLOYEES.
                MOVE SALARY-TOTAL TO SL-SALARY-TOTAL.
                WRITE PRINT-LINE FROM SUMMARY-LINE AFTER 3.

        A900-TERMINATION.
            PERFORM A600-DEPARTMENT-BREAK.
            PERFORM A650-DIVISION-BREAK.
            PERFORM A800-WRITE-SUMMARY-LINE.
            CLOSE EMPLOYEE-FILE
            DEPT-DIVISION-LIST.
```

A.1.2 Employee Data File

```
JONES BRIAN      6463 FRENCHMENS DRIVE,ALEX,VA,  156780225 1 CS 123456 01 F
SMITH DOUG      1234 ANY WHERE ,MCLN,VA, 123450999 1 CS009999 02 C
JOHN DOE        9999 MY STREET ,WHTH,VA,999999998 2 ST019899 03 H
JANE DOE         7777 GOOD STREET ,NATICK,MA,777000555 3 SE567890 02 F
```

A.1.3 Tax Report Listing

```
EMPLOYEE NAME     EMP NUMBER    STATUS  DEPARTMENT  DEP   SALARY       TOT-TAX
JONES BRIAN        156780225      F      CS          1   $123,456.00  $43,669.16
SMITH DOUG         123450999      C      CS          2   $ 9,999.00   $ .00
                   2 EMPLOYEES    $***133,455.00 $*43,669.16
                   ****** DIVISION TOTAL ******  2 EMPLOYEES  $***133,455
JOHN DOE           999999998      H      ST          3   $ 19,899.00  $2,854.80
                   1 EMPLOYEES           $****19,899.00       $**2,854.80
 ****** DIVISION TOTAL ****** 3 EMPLOYEES        $****19,899
JANE DOE           777000555      F      SE          2   $567,890.00  $102,890.40
```

```
                        1 EMPLOYEES                $***567,890.00          $102,890.40
****** DIVISION TOTAL ****** 4 EMPLOYEES          $***567,890
COMPANY TOTAL       ***** 4 EMPLOYEES$***721,244.00
```

A.2 ADA PACKAGE SPECIFICATIONS

A.2.1 ss_s.ada

```
--------------
-- Abstraction
--------------
package Social_Security is

  type Number is private;

  Default_Separator : constant Character := ' ';

  Invalid_Number : exception;
  function Construct (Part1 : in     Natural;
                      Part2 : in     Natural;
                      Part3 : in     Natural) return Number;

  function Image (Self      : in Number;
                  Separator : in Character := Default_Separator)
    return String;

private
  type Number is new String(1..11);
end Social_Security;
```

A.2.2 employee_s.ada

```
with Social_Security;
with ADAR_Comp;
--=======
-- Class:
--=======
package Employee is
  type Class is private;
  --=============
  -- Attributes:
  --=============
  type Name       is new String (1..25);
  type Number     is new Social_Security.Number;
  type Department is (Unknown,
                      Computer_Science,
                      Science_and_Technology,
                      Systems_Evaluation);
  type Status     is (Salaried, Hourly, Consultant);
```

```
 type Money      is new ADAR_Comp.Decimal (Precision => 9, Scale => 2);
 --====================
 -- Object Management:
 --====================
 -- Without a Constructor, this type cannot be used. Look at subclasses
 -- to see how to construct objects. In Ada 95, Class can be an
 -- abstract type!
 -- ============================
 -- Attribute access operations:
 -- ============================
 procedure Change (Self       : in Class;
                   Emp_Name   : in Name;
                   SS_Number  : in Number);
 -- Overloading is a form of (ad-hoc) polymorphism:
 procedure Change (Self : in Class; D      : in Department);
 procedure Change (Self : in Class; Salary : in Money);

 function Emp_Name       (Self : in Class) return Name;
 function SS_Number      (Self : in Class) return Number;
 function Emp_Status     (Self : in Class) return Status;
 function Emp_Department (Self : in Class) return Department;
 function Emp_Salary     (Self : in Class) return Money;

 --=============
 -- Operations:
 --=============
 package Abstract is
   function Net_Pay       (Self : in Class) return Money;
   function Send_Check_To (Self : in Class) return String;
 end Abstract;

----------------------------------------
-- For Child packages only (see Ada 95):
-- Start private section here when migrating to Ada 95!

  -- Status is used as the tag to simulate polymorphism.
  type Tag is new Status;
  type Structure (Tagged : Tag) is private;

private
  -- The discriminant, Tagged, is used to simulate runtime polymorphism.
  -- This should be replaced in Ada 9X with the corresponding tagged record
  -- declaration.
  type Structure (Tagged : Tag) is
    record
      Emp_Name       : Name;
      SS_Number      : Number;
      Emp_Department : Department;
      Emp_Salary     : Money;
```

```
   end record;

  type Class is access Structure;

end Employee;
```

A.2.3 employee_consulting_s.ada

```
with Employee;

--==========
-- Subclass
--==========
package Employee_Consulting is
  type Class is new Employee.Class; -- Inheritance

  --====================
  -- Object Management
  --====================
  procedure Initialize (Object : in out Class);

  --================================
  -- New attribute access operations
  --================================
  procedure Set_Mail_Address (Self : in Class; MA : in String);
  function  Mail_Address     (Self : in Class) return  String;

  --================
  -- New operations
  --================
  -- The Dispatching methods declared in the parent class must
  -- be defined for each subclass and the dispatching method itself
  -- updated to invoke the correct subclass method.
  function Net_Pay       (Self : in Class) return Employee.Money;
  function Send_Check_To (Self : in Class) return String;

-----------------------------------------------------------------------
-- For dispatching only. See implementation of Parent Class. Remove
-- in Ada 95.
  Unique_Tag : constant Employee.Tag := Employee.Consultant;

private
  -- A quick and dirty way to deal with unconstrained attributes:
  type Mailing_Address is access String;

  -- The subclass structure must keep the parent structure intact
  -- while appending additional data. This implementation works for
  -- most compilers:
  type Structure is
```

```
   record
     Parent : Employee.Structure(Unique_Tag);
     MA     : Mailing_Address;
   end record;
end Employee_Consulting;
```

A.2.4 employee_taxable_s.ada

```
with Employee;

--==========
-- Subclass
--==========
-- This package combines two subclasses, something Ada can do more
-- conveniently than other languages:
package Employee_Taxable is
  type Class is new Employee.Class;

  --=============
  -- Attributes:
  --=============
  -- Tax uses the already defined Employee.Money type.
  type Deduction is range 0..12;

  --====================
  -- Object management:
  --====================
  procedure Initialize_Hourly   (Object : in out Class);
  procedure Initialize_Salaried (Object : in out Class);

  --============================
  -- New attribute operations:
  --============================
  procedure Change (Self : in Class; Tax : in Employee.Money);
  procedure Change (Self : in Class; D   : in Deduction);

  function Tax        (Self : in Class) return Employee.Money;
  function Deductions (Self : in Class) return Deduction;

  --=================
  -- New operations:
  --=================
  function Net_Pay      (Self : in Class) return Employee.Money;
  function Send_Check_To (Self : in Class) return String;

-------------------------------------------------------------
  Unique_Hourly_Tag   : constant Employee.Tag := Employee.Hourly;
  Unique_Salaried_Tag : constant Employee.Tag := Employee.Salaried;
```

```
private
  -- The two different structures are identical except for the tag:
  type Structure_Hourly is   -- new Employee.Structure with
    record
      Parent : Employee.Structure(Unique_Hourly_Tag);
      Tax    : Employee.Money;
      D      : Deduction;
    end record;
  type Structure_Salaried is -- new Employee.Structure with
    record
      Parent : Employee.Structure(Unique_Salaried_Tag);
      Tax    : Employee.Money;
      D      : Deduction;
    end record;
end Employee_Taxable;
```

A.2.5 employee_file_s.ada

```
with Employee; -- An associated class

--=======
-- Class
--=======
package Employee_File is
  type Class is limited private;

  --====================
  -- Object management:
  --====================
  -- The Open procedure provides the constructor method:
  Unable_to_Open_File : exception;
  procedure Open  (Self      : in out Class;
                   Path_Name : in     String);

  -- The Close procedure provides the destructor method:
  procedure Close (Self : in out Class);

  --=============
  -- Operations:
  --=============
  Unable_to_Read_File      : exception;
  Attempt_to_Read_Past_EOF : exception;
  function Get_Next (Self   : in Class) return Employee.Class;

  function End_of_File (Self : in Class) return Boolean;

private
  -- When Structure is not visible, cannot inherit from this class. A tag
```

```
  -- is not needed, either.
  type Structure;

  type Class is access Structure;
end Employee_File;
```

A.2.6 check_s.ada

```
with Employee; -- an associated class
with Calendar; -- used for the date attribute

--=======
-- Class
--=======
package Check is
  type Class is private;

  --====================
  -- Object management:
  --====================
  -- This class has an association with Employee.Class which is
  -- implemented one way.
  function Construct (Pays   : in Employee.Class;
                      Number : in Natural;
                      Date   : in Calendar.Time := Calendar.Clock)
    return Class;

  --============
  -- Operations
  --============
  procedure Print (Self : in Class);

private
  type Structure is
    record
      Pays   : Employee.Class;
      Number : Natural;
      Date   : Calendar.Time;
    end record;

  type Class is access Structure;

end Check;
```

A.2.7 payroll.ada

```
with Employee_File;
```

```
with Check;
with Tax_Calculation;

with Text_IO; -- For User Interface

--======
-- Class
--======
-- This is a control class, most easily implemented as a procedure,
-- although a package could be used in preparation for a more
-- sophisticated user interface (such as X-windows callbacks).
procedure Payroll is
  -- There is only one payroll, therefore a type definition is not needed.

  --===========
  -- Attributes
  --===========
  DB          : Employee_File.Class;
  Check_Number : Natural;
  Report_File_Name : constant String := "PRINTER.DAT";

  -- User-Interface
  Input_Buffer : String (1..80);
  Input_Length : Natural;

  --===========
  -- Operations
  --===========
begin
  Generate_Report:
    begin
      Tax_Calculation;
    end Generate_Report;

  Set_File_Name:
    begin
      Employee_File.Open (DB, Report_File_Name);
    end Set_File_Name;

  Set_Starting_Check_Number:
    begin
      Text_IO.Put_Line ("Enter starting check number:");
      Text_IO.Get_Line (Input_Buffer, Input_Length);
      Check_Number := Natural'Value (Input_Buffer (1..Input_Length));
    end Set_Starting_Check_Number;

  Print_Payroll:
    begin
```

```
    while not Employee_File.End_of_File (DB) loop
     begin
      Check.Print (Check.Construct (Employee_File.Get_Next (DB),
                                    Check_Number));
      Check_Number := Check_Number + 1;
     exception
      when Employee_File.Attempt_to_Read_Past_EOF =>
       exit;
      when others =>
       null;
     end;
    end loop;

    Employee_File.Close (DB);
  end Print_Payroll;

end Payroll;
```

A.3 ADA PACKAGE BODIES

A.3.1 ss_b.ada

```
package body Social_Security is
 ---------------------
 Operation definitions
 ----------------------
 function Fixed_Image (N      : in Natural;
                       Length : in Natural) return String is
  Result : String(1..Length) := String'(1..Length => '0');
  Image  : constant String := Natural'Image(N);
  L      : constant Natural := Image'Length;
 begin
  Result(2+Length-L..Length) := Image(Image'First+1..Image'Last);
  return Result;
 end Fixed_Image;
 ----------
 function Construct (Part1 : in     Natural;
                     Part2 : in     Natural;
                     Part3 : in     Natural) return Number is
 begin
  if Part1 in 0..999  and
     Part2 in 0..99   and
     Part3 in 0..9999 then
    return Number (Fixed_Image (Part1,3) & '-' &
                   Fixed_Image (Part2,2) & '-' &
                   Fixed_Image (Part3,4)
                  );
  else
    raise Invalid_Number;
```

```
  end if;
 end Construct;
 ----------
 function Image (Self      : in Number;
                 Separator : in Character := Default_Separator)
  return String is
 begin
  if Separator = Default_Separator then
   return String(Self);
  else

   Convert_Delimeter:
   declare
    Str : String(1..Self'Length) := String(Self);
   begin
    Str(4) := Separator;
    Str(7) := Separator;
    return Str;
   end Convert_Delimeter;

  end if;
 end Image;

end Social_Security;
```

A.3.2 employee_b.ada

```
package body Employee is
 ---------------------
 Operation definitions
 ----------------------
 procedure Change (Self       : in Class;
                   Emp_Name   : in Name;
                   SS_Number  : in Number) is
 begin
  Self.Emp_Name  := Emp_Name;
  Self.SS_Number := SS_Number;
 end;
 ----------
 procedure Change (Self : in    Class; D : in    Department) is
 begin
  Self.Emp_Department := D;
 end;
 ----------
 procedure Change (Self : in    Class; Salary : in    Money) is
 begin
  Self.Emp_Salary := Salary;
 end;
```

```
  ----------
  function Emp_Name  (Self : in Class) return Name is
  begin
    return Self.Emp_Name;
  end;
  ----------
  function SS_Number (Self : in Class) return Number is
  begin
    return Self.SS_Number;
  end;
  ----------
  function Emp_Status     (Self : in Class) return Status is
  begin
    return Status(Self.Tagged);
  end;
  ----------
  function Emp_Department (Self : in Class) return Department is
  begin
    return Self.Emp_Department;
  end;
  ----------
  function Emp_Salary     (Self : in Class) return Money is
  begin
    return Self.Emp_Salary;
  end;
  -- Dispatching operations can be separate to make updates easier
  -- and to localize the context clauses to the operations that use them.
  package body Abstract is separate;

end Employee;
```

A.3.3 employee_consulting_b.ada

```
with Unchecked_Conversion; -- For simulating inheritance
with Ada; -- .Tags

package body Employee_Consulting is
  --=========================
  -- Subclass Implementation
  --=========================
  Data type definition
  ---------------------
  type Child_Pointer is access Structure; -- of the Child.
  ---------------------
  Operation definitions
  ----------------------
  function Narrow (Parent_Pointer : in Class) return Child_Pointer is

    function Convert_Pointer is
```

```
    new Unchecked_Conversion (Source => Class,
                              Target => Child_Pointer);

   Result : constant Child_Pointer
         := Convert_Pointer (Parent_Pointer);
   use Employee;
  begin
   if Result.Parent.Tagged = Unique_Tag then -- of this Child
     return Result;
   else
     raise Ada.Tags.Tag_Error;
   end if;
  end Narrow;
  ----------
  procedure Initialize (Object : in out Class) is
   function Convert_Pointer is
     new Unchecked_Conversion (Child_Pointer, Class);
  begin
   Object := Convert_Pointer (new Structure); -- of the Child
  end;
  ----------
  function Mail_Address (Self : in Class) return String is
   P : constant Child_Pointer := Narrow(Self);
  begin
   if P.MA = null then  return "Hold";
   else                 return P.MA.all;
   end if;
  end;
  ----------
  procedure Set_Mail_Address (Self : in     Class; MA : in String) is
  begin
   Narrow(Self).MA := new String'(MA);
  end;
  ----------
  function Net_Pay      (Self : in Class) return Employee.Money is
  begin
   return Emp_Salary(Self);
  end;
  ----------
  function Send_Check_To (Self : in Class) return String is
  begin
   return Mail_Address(Self);
  end;

 end Employee_Consulting;
```

A.3.4 employee_taxable_b.ada

```
with Unchecked_Conversion; -- For simulating inheritance
with Ada; --.Tags

package body Employee_Taxable is
 --=========================
 -- Subclass Implementation
 --=========================
 Data type definitions
 ---------------------
 type Pointer_H is access Structure_Hourly;
 type Pointer_S is access Structure_Salaried;
 ---------------------
 Operation definitions
 ---------------------
 -- Narrow arbitrarily uses Pointer_H, it could use Pointer_S:
 function Narrow (Parent_Pointer : in Class) return Pointer_H is
  function Convert_Pointer is
   new Unchecked_Conversion (Source => Class,
                             Target => Pointer_H);
  Result : constant Pointer_H := Convert_Pointer(Parent_Pointer);
  use Employee; -- for "=" operator
 begin
  if Result.Parent.Tagged = Hourly or
    Result.Parent.Tagged = Salaried then
   return Result;
  else
   raise Ada.Tags.Tag_Error;
  end if;
 end Narrow;
-----------
 procedure Initialize_Hourly (Object : in out Class) is

  function Convert_Pointer is
   new Unchecked_Conversion (Pointer_H, Class);

 begin
  Object := Convert_Pointer (new Structure_Hourly);
 end;
-----------
 procedure Initialize_Salaried (Object : in out Class) is

  function Convert_Pointer is
   new Unchecked_Conversion (Pointer_S, Class);
 begin
  Object := Convert_Pointer (new Structure_Salaried);
 end;
```

```
 -----------
  procedure Change (Self : in Class; Tax : in     Employee.Money) is
  begin
    Narrow(Self).Tax := Tax;
  end;
 -----------
  procedure Change (Self : in Class; D   : in Deduction) is
  begin
    Narrow(Self).D := D;
  end;
 -----------
  function Tax       (Self : in Class) return Employee.Money is
  begin
    return Narrow(Self).Tax;
  end Tax;
 -----------
  function Deductions (Self : in Class) return Deduction is
  begin
    return Narrow(Self).D;
  end Deductions;
 -----------
  function Net_Pay      (Self : in Class) return Employee.Money is
    Result : Employee.Money := Emp_Salary(Self);
    use Employee;
  begin
    Decrement (Result, Tax(Self), Rounded => True);
    return Result;
  end;
 -----------
  function Send_Check_To (Self : in Class) return String is
  begin
    return Employee.Department'Image (Emp_Department(Self));
  end;

end Employee_Taxable;
```

A.3.5 empolyee_abstract.ada

```
with Employee_Taxable;
with Employee_Consulting;

separate (Employee)
package body Abstract is
 ----------------------
  Operation definitions
 ----------------------
  function Net_Pay (Self : in Class) return Money is
  begin -- Dispatching
    case Self.Tagged is
```

```
    when Hourly | Salaried =>
      return Employee_Taxable.Net_Pay
        (Employee_Taxable.Class   (Self));

    when Consultant       =>
      return Employee_Consulting.Net_Pay
        (Employee_Consulting.Class(Self));
    -- Add additional children here
    -- No others clause! This is an abstract operation!
  end case;
 end;
-----------
 function Send_Check_To (Self : in Class) return String is
 begin
   case Self.Tagged is
     when Hourly | Salaried =>
       return Employee_Taxable.Send_Check_To
            (Employee_Taxable.Class   (Self));

     when Consultant       =>
       return Employee_Consulting.Send_Check_To
            (Employee_Consulting.Class(Self));

     -- Add additional children here
     -- No others clause! This is an abstract operation!

   end case;
 end;

end Abstract;
```

A.3.6 employee_file_b.ada

```
with Employee_Taxable;
with Employee_Consulting;

with ADAR_Comp;
with Sequential_IO;

-- Most of this file involves parsing an ASCII text file.
package body Employee_File is

-- COBOL specification of data file format:
--01     DETAIL-LINE.
--       02     DL-NAME             PIC     X(25).
--       02     FILLER              PIC     X(8)    VALUE   SPACES.
--       02     DL-EMPLOYEE-NUMBER     PIC     X(9).
--       02     FILLER              PIC     X(9)    VALUE   SPACES.
--       02     DL-STATUS           PIC     X.
```

```
--      02    FILLER              PIC    X(7)   VALUE   SPACES.
--      02    FILLER              PIC    X(6)   VALUE   SPACES.
--      02    DL-DEPARTMENT       PIC    X(2).
--      02    FILLER              PIC    X(5)   VALUE   SPACES.
--      02    DL-DEPENDENT        PIC    Z9.
--      02    FILLER              PIC    X(12)  VALUE   SPACES.
--      02    DL-SALARY           PIC    $ZZZ,ZZZ.99.
--      02    FILLER              PIC    X(8)   VALUE   SPACES.
--      02    DL-TAX              PIC    $ZZZ,ZZZ.99.
 ----------------------
 Data type definitions
 ----------------------
 type Detail_Line is
  record
    Emp_Name      : String (1..25);
    Filler_1      : String (1..8);
    Emp_Number    : String (1..9);
    Filler_2      : String (1..9);
    Emp_Status    : Character;
    Filler_3      : String (1..13);
    Emp_Department : String (1..2);
    Filler_4      : String (1..5);
    Emp_Deductions : String (1..2);
    Filler_5      : String (1..12);
    Emp_Salary    : String (1..11);
    Filler_6      : String (1..8);
    Tax           : String (1..11);
    Filler_7      : String (1..16);
  end record;

type Status_Conversion is array (Character) of Employee.Status;
Convert_Status : constant Status_Conversion
   := Status_Conversion'('s' | 'S' |'f' | 'F' => Employee.Salaried,
                         'h' | 'H' => Employee.Hourly,
                         'c' | 'C' => Employee.Consultant,
                         others    => Employee.Hourly);

type Department_Code is (CS,ST,SE);
type Conversion is array (Department_Code) of Employee.Department;
Department_Convert : constant Conversion
            := (CS => Employee.Computer_Science,
                ST => Employee.Science_and_Technology,
                SE => Employee.Systems_Evaluation);

package File_Operations is
  new Sequential_IO (Detail_Line);

type Structure is
```

```
  record
    File : File_Operations.File_Type;
  end record;
---------------------
Operation definitions
---------------------
procedure Open  (Self      : in out Class;
                 Path_Name : in     String) is
begin
  if    Self = null       then Self := new Structure;
  elsif File_Operations.Is_Open (Self.File) then
    File_Operations.Close (Self.File);
  end if;

  File_Operations.Open (File => Self.File,
                        Mode => File_Operations.In_File,
                        Name => Path_Name);
exception
  when others =>
    raise Unable_to_Open_File;
end Open;
----------
procedure Close (Self : in out Class) is
begin
  if Self /= null then
      File_Operations.Close (Self.File);
      -- Deallocate Self -- TBD
  end if;
end Close;
----------
function Salary_Value (S : in String) return Employee.Money is
  Parse_S : String(1..S'Length) := S;
  Result  : Employee.Money;
begin
  Zero_Leading:
    for I in Parse_S'Range loop
      case Parse_S(I) is
        when '$' | '*' | ' ' => Parse_S(I)    := '0';
        when ','             => Parse_S(2..I) := Parse_S(1..I-1);
                                Parse_S(1)    := ' ';
        when others          => null;
      end case;
    end loop Zero_Leading;

  Employee.Move (Parse_S, Result);
  return Result;
end Salary_Value;
----------
```

```
function Construct (Tag  : in Employee.Status;
                    Name : in Employee.Name;
                    SS   : in Employee.Number)
  return Employee.Class is
  Result : Employee.Class;
  use Employee;
begin
  case Tag is
    when Consultant => Employee_Consulting.Initialize
      (Employee_Consulting.Class(Result));
    when Salaried   => Employee_Taxable.Initialize_Hourly
      (Employee_Taxable.Class(Result));
    when Hourly     => Employee_Taxable.Initialize_Salaried
      (Employee_Taxable.Class(Result));
  end case;
  Employee.Change (Result, Name, SS);

  return Result;

end;
----------
function Get_Next  (Self   : in Class) return Employee.Class is
  Data_Record : Detail_Line;
begin
  if File_Operations.End_of_File (Self.File) then
    raise Attempt_to_Read_Past_EOF;
  end if;

  File_Operations.Read (Self.File, Data_Record);

  Parse_Data_Record:
  declare
    Result : constant Employee.Class := Construct (
             Tag  => Convert_Status(Data_Record.Emp_Status),
             Name => Employee.Name(Data_Record.Emp_Name),
             SS   => Employee.Construct (
               Part1 => Natural'Value (Data_Record.Emp_Number(1..3)),
               Part2 => Natural'Value (Data_Record.Emp_Number(4..5)),
               Part3 => Natural'Value (Data_Record.Emp_Number(6..9))));
  begin
    Employee.Change (Result, Salary_Value(Data_Record.Emp_Salary));

    Determine_Department:
    declare
      D : Department_Code;
    begin
      D := Department_Code'Value (Data_Record.Emp_Department);
      Employee.Change (Result, Department_Convert (D));
```

```
    exception
      when others => Employee.Change (Result, Employee.Unknown);
    end Determine_Department;

    case Employee.Emp_Status(Result) is
      when Employee.Salaried | Employee.Hourly =>
        Employee_Taxable.Change (Self => Employee_Taxable.Class(Result),
                                 Tax  => Salary_Value (Data_Record.Tax));
        Employee_Taxable.Change (Employee_Taxable.Class(Result),
          Employee_Taxable.Deduction'Value(Data_Record.Emp_Deductions));
      when others =>
        null;
    end case;

    return Result;
  end Parse_Data_Record;

 exception
  when others =>
    raise Unable_to_Read_File;
 end Get_Next;
 ----------
 function End_of_File (Self : in Class) return Boolean is
 begin
  return Self = null or else
        not File_Operations.Is_Open (Self.File) or else
        File_Operations.End_Of_File (Self.File);
 end End_of_File;

end Employee_File;
```

A.3.7 check_b.ada

```
with Text_IO;   -- To print the check
with ADAR_Comp; -- Format money

package body Check is
 ---------------------
 Operation definitions
 ----------------------
 function Construct (Pays   : in Employee.Class;
                     Number : in Natural;
                     Date   : in Calendar.Time := Calendar.Clock)
  return Class is
  Result : Class := new Structure;
 begin
  Result.Pays   := Pays;
  Result.Number := Number;
```

```
   Result.Date   := Date;

   return Result;
  end Construct;
  ----------
  function Date_Image (Date : Calendar.Time) return String is
   use Calendar;
  begin
   return Month_Number'Image(Month(Date)) & '/' &
         Day_Number'Image  (Day  (Date)) & '/' &
         Year_Number'Image (Year (Date));
  end;
  ----------
  procedure Print (Self : in Class) is
  begin
   Text_IO.New_Line;
   Text_IO.Put_Line(Integer'Image(Self.Number) &
     String'(1..10 => ' ') & Date_Image(Self.Date));
   Text_IO.New_Line;

   Text_IO.Put_Line (String(Employee.Emp_Name(Self.Pays)) &
       String'(1..5 => ' ') &
      '$' & Employee.Image(Employee.Abstract.Net_Pay(Self.Pays)));

   Text_IO.New_Line;
   Text_IO.Put_Line ("Send Check to:" &
                Employee.Abstract.Send_Check_To(Self.Pays));
   Text_IO.New_Line;
  end;

end Check;
```

LIST OF REFERENCES

[ANSI95] American National Standards Institute, ANSI/ISO/IEC-8652:1995, *Ada 95 Reference Manual: The Language, The Standard Libraries*, New York, NY, January 1995.

[BLSM93] Standard Systems Center/XON, *Base-Level System Modernization (BLSM): A Strategy for the Future*, Maxwell AFB, Gunter Annex, AL, November 1, 1993.

[BOO94] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1994.

[BROD93] M. L. Brodie and M. Stonebraker, *Darwin: On the Incremental Migration of Legacy Information Systems*, Technical Report TR-0222-10-92-165, GTE Laboratories, Inc., Waltham, MA, 1993.

[CDYD91] P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd edition, Yourdon Press, Englewood Cliffs, NJ, 1991.

[CHAS90] G. Chastek, M. H. Graham, and G. Zelesnik, *The SQL Ada Model Description Language—SAMeDL*, Technical Report CMU/SEI-90-TR-26, Software Engineering Institute, Pittsburgh, PA, 1990.

[CIM94] Center for Information Management, *Center for Information Management Software Systems Reengineering Process Model, Version 2.0*, draft, Defense Information Systems Agency, Joint Interoperability Engineering Organization, Fairfax, VA, September 1994.

[DATE75] C. J. Date, *An Introduction to Database Systems*, 1st edition, Addison-Wesley, Reading, MA, 1975.

[DIET89] W. C. Dietrick, Jr., L. R. Nackman, and F. Gracer. “Saving a Legacy with Objects,” *OOPSLA’89 Proceedings*, 1989, pp. 77-83.

[DOD92] Department of Defense, DoD 8020.1-M, *Functional Management Process for Implementing the Information Management Program of the Department of Defense*, draft, August 1992.

[DOD93] Department of Defense, DoD Directive 8120.1, *Life-Cycle Management (LCM) of Automated Information Systems (AISs)*, January 14, 1993.

[DOD94a] Department of Defense, MIL-STD-498 Software Development and Documentation, December 5, 1994.

[DOD94b] Department of Defense, Data Item Descriptions (DIDs) for MIL-STD-498.

[DON87] J. E. D. Donaho and G. K. Davis, "Ada Embedded SQL: The Options," *ACM Ada Letters*, Vol. VII, No. 3, May/June 1987.

[GRAH89] M. H. Graham, *Guidelines for the Use of the SAME*, Technical Report CMU/SEI-89-TR-16, Ada 228027, Software Engineering Institute, Carnegie Mellon University, PA, May 1989.

[HUTT94] A. T. F. Hutt, ed., *Object Analysis and Design: Description of Methods*, John Wiley & Sons, New York, NY, 1994.

[IDA95a] B. A. Haugh, M. C. Frame, and K. Jordan, *An Object-Oriented Development Process for Department of Defense Information Systems*, IDA Paper P-3142, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95b] D. Smith, B. Haugh, and K. Jordan, *Object-Oriented Programming Strategies for Ada*, IDA Paper P-3143, Institute for Defense Analyses, Alexandria, VA, July 1995.

[LOO94] C. Loosley, "A Three-Tier Solution: Achieving Data Integrity by Using Objects and Relational DBMSs Together," *Database Programming and Design*, February 1994, pp. 23-25.

[MCC90] L. S. McCoy, "Bindings and Ada," *ACM Ada Letters*, Vol. X, No. 8, November/December 1990.

[NACK86] L. R. Nackman et al., "AML/X: A Programming Language for Design and Manufacturing," *Proceedings of the Fall Joint Computer Conference*, November 1986, pp. 145-159.

[NEL91] M. Nelson, "An Object-Oriented Tower of Babel," *ACM OOPS Messenger*, Vol. 2, No. 3, July 1991.

[ORCL92] Oracle Corporation, “Pro Ada Precomplier,” Programmer’s Guide Version 1.4, 1992.

[RUMB91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[SEI91] Software Engineering Institute, *Rationale for SQL Ada Module Description Language SAMEDL*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1991.

[TAYL92] D. A. Taylor, *Object-Oriented Information Systems: Planning and Implementation*, John Wiley & Sons, New York, NY, 1992.

[WEG90] P. Wegner, “Concepts and Paradigms of Object-Oriented Programming,” *OOPS Messenger* Vol. 1/No. 1, August 1990.

[WESL80] M. A. Wesley, T. Lozano-Perez, L. I. Lieberman, M. A. Lavin, and D. Grossman, “A Geometric Modeling System for Automated Mechanical Assembly,” *IBM Journal of Research and Development*, Vol. 24, No. 1, January 1980, pp. 64-74.

[WIR90] R. Wirfs-Brock, B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[WOLF87] R. Wolfe, M. Wesley, J. Kyle Jr., F. Gracer, and W. Fitzgerald, “Solid Modeling for Production Design,” *IBM Journal of Research and Development*, Vol. 31, No. 3, May 1987, pp. 277-295.

GLOSSARY

Words used in the definition of a glossary term and that are defined elsewhere are in **bold**.

Abstraction — Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties [RUMB91].

AIS Program — A directed and funded AIS effort, to include all **migration** systems, that is designed to provide a new or improved capability in response to a validated need [DOD93].

Architecture — The organizational structure of a system or **CSCI**, identifying its components, their interfaces, and a concept of execution among them [DOD94a].

Automated Information System (AIS) — A combination of **computer hardware** and computer **software**, data and/or telecommunications that performs functions such as collecting, processing, transmitting, and displaying information. Excluded are computer resources, both hardware and software, that are physically part of, dedicated to, or essential in real time to the mission performance of **weapon systems**; used for weapon system specialized training, simulation, diagnostic test and maintenance, or calibration; or used for research and development of weapon systems [DOD93]. However, as used here, AISs include systems for C2I, C3I, and C4I, even though they may be essential in real time to mission performance.

Class — A class can be defined as a description of similar **objects**, like a template or cookie cutter [NEL91]. The class of an object is the definition or description of those attributes and behaviors of interest.

CRC Cards — **Class-Responsibility**-Collaborator Cards. CRC cards are pieces of paper divided into three areas: the **class** name and the purpose of the class, the responsibilities of the class, and the collaborators of the class. CRC cards are intended to be used to iteratively simulate different scenarios of using the system to get a better understanding of its nature [HUTT94, p. 192].

Collaboration — A request from a client to a server in fulfillment of a client's **responsibilities** [HUTT94, p. 192].

Commercial-off-the-Shelf (COTS) — Commercial items that require no unique government modifications or maintenance over the life cycle of the product to meet the needs of the procuring agency [DOD93].

Computer Hardware — Devices capable of accepting and storing computer data, executing a systematic sequence of **operations** and computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions [DOD94a]

Computer Program — A combination of computer instructions and data definitions that enable **computer hardware** to perform computational or control functions.

Computer Software Configuration Item (CSCI) — An aggregation of **software** that satisfies an end use function and is designated for separate configuration management by the acquirer. CSCIs are selected based on tradeoffs among software function, size, host or target computers, developer, support concept, plans for reuse, criticality, and interface considerations need to be separately documented and controlled, and other factors.

Contract — The list of requests that a client **class** can make of a server class. Both must fulfill the contract: the client by making only those requests the contract specifies, and the server by responding appropriately to those requests [HUTT94, p. 192].

Database — A collection of related data stored in one or more computerized files in a manner that can be accessed by users or **computer programs** via a **database management system** [DOD94a].

Database Management System	An integrated set of **computer programs** that provide the capabilities needed to establish, modify, make available, and maintain the integrity of a database [DOD94b].
Encapsulation	. . . (also **information hiding**) consists of separating the external aspects of an **object**, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects [RUMB91]. The act of grouping into a single object both data and the **operation** that affects that data [WIR90].
Framework	Collection of **class** libraries, generics, design, scenario models, documentation, etc., that serves as a platform to build applications.
Government-off-the-Shelf (GOTS)	Products for which the Government owns the data rights, that are authorized to be transferred to other DoD or Government customers, and that require no unique modifications or maintenance over the life cycle of the product [DOD93b].
Inheritance	Inheritance is the sharing of attributes and **operations** among **classes** based on a hierarchical relationship [RUMB91]. Subclasses of a class inherit the operations of the parent class and may add new operations and new instance variables. Inheritance allows us to reuse the behavior of a class in the definition of new classes [WEG90].
Information Hiding	Making the internal data and methods inaccessible by separating the external aspects of an **object** from the internal (hidden) implementation details of the object.
Information System	See **Automated Information System (AIS)**.
Legacy System	Any currently operating automated system that incorporates obsolete computer technology, such as proprietary hardware, closed systems, “stovepipe” design, or obsolete programming languages or **database** systems.
Life-Cycle Management (LCM)	A management process, applied throughout the life of an **AIS**, that bases all programmatic decisions on the anticipated mis-

	sion-related and economic benefits derived over the life of the AIS [DOD93].
Message	Mechanism by which **objects** in an OO system request **services** of each other. Sometimes this is used as a synonym for **operation**.
Migration	The transition of support and **operations** of software functionality from a **legacy system** to a **migration system**.
Migration System	An existing **AIS**, or a planned and approved AIS, that has been officially designated to support standard processes for a functional activity applicable DoD-wide or DoD Component-wide [DOD93]. Ordinarily, an AIS that has been designated to assume the functionality of a legacy AIS.
Monomorphism	A concept in type theory, according to which a name (such as a variable declaration) may only denote **objects** of the same **class**.
Object	A combination of state and a set of methods that explicitly embodies an **abstraction** characterized by the behavior of relevant requests. An **object** is an instance of an implementation and an interface. An object models a real-world entity (such as a person, place, thing, or concept), and it is implemented as a computational entity that encapsulates state and **operations** (internally implemented as data and methods) and responds to requestor **services**.
Object-Oriented Analysis	A method of analysis in which requirements are examined from the perspective of the **classes** and **objects** found in the vocabulary of the problem domain [BOO94].
Object-Oriented Decomposition	The process of breaking a system into parts, each of which represents some **class** or **object** from the problem domain [BOO94].
Object-Oriented Design	A method of design encompassing the process of OO decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design

[BOO94].

Object-Oriented Programming — A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some **class**, and whose classes are members of a hierarchy of classes united via **inheritance** relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature, which is encouraged by the existence of dynamic binding and **polymorphism** [BOO94].

Object-Oriented Technology (OOT) — OOT consists of a set of methodologies and tools for developing and maintaining **software systems** using software **objects** composed of encapsulated data and **operations** as the central paradigm.

Object Request Broker (ORB) — Program that provides a location and implementation independent mechanism for passing a **message** from one object to another.

Operation — A specific behavior that an **object** exhibits, implemented as a procedure contained within the object.

Polymorphism — The same **operation** may behave differently on different **classes** [RUMB91].

Reengineering — The process of examining and altering an existing system to reconstitute it in a new form. May include reverse engineering (analyzing a system and producing a representation at a higher level of **abstraction**, such as design from code), restructuring (transforming a system from one representation to another at the same level of abstraction), redocumentation (analyzing a system and producing user or support documentation), forward engineering (using **software** products derived from an existing system, together with new **requirements**, to produce a new system), retargeting (transforming a system to install it on a different target system), and translation (transforming source code from one language to another or from one version of a language to another) [DOD94a].

Requirement (1) characteristic that a system or **CSCI** must possess in order to be acceptable to the acquirer. (2) A mandatory statement in this standard or another portion of the **contract** [DOD94a].

Responsibility A **contract** that a **class** must support, intended to convey a sense of the purpose of the class and its place in the system [HUTT94, p. 192].

Service A service is a specific behavior that an **object** is responsible for exhibiting [CDYD91].

Software **Computer programs** and computer **databases**. **Note:** Although some definitions of software includes documentation, MIL-STD-498 limits the scope of this term to computer programs and computer databases in accordance with Defense Federal Acquisition Regulation Supplement 227.401 [DOD94a].

Software Development A set of activities that results in **software** products. Software development may include new development, modification, reuse, **reengineering**, or any other activities that result in software products [DOD94a].

Software Engineering In general usage, a synonym for **software development**. As used in this standard [MIL-STD-498], a subset of software development consisting of all activities except qualification testing. The standard makes this distinction for the sole purpose of giving separate names to the **software engineering** and software test environments [DOD94a].

Software Engineering Environment The facilities, hardware, **software**, firmware, procedures, and documentation needed to perform **software engineering**. Elements may include but are not limited to computer-assisted software engineering (CASE) tools, compilers, assemblers, linkers, loaders, operating systems, debuggers, simulators, emulators, documentation tools, and **database management systems**.

Software System A system consisting solely of **software** and possibly the computer equipment on which the software operates [DOD94a].

Weapon System Items that can be used directly by the Armed Forces to carry out

combat missions and that cost more than 100,000 dollars or for which the eventual total procurement cost is more than 10 million dollars. That term does not include commercial items sold in substantial quantities to the general public (Section 2403 of 10 U.S.C., reference (bb)) [DOD93].

LIST OF ACRONYMS

ADAR	Ada Decimal Arithmetic and Representatives
AIS	Automated Information Systems
AP	Application Program
API	Application Programming Interface
CICS	Customer Information Control System
CORBA	Common Object Request Broker Architecture
DBMS	Database Management Systems
DBS	Database Server
DEC	Digital Equipment Corporation
DISA	Defense Information Systems Agency
DoD	Department of Defense
GDP	Geometric Design Processor
GUI	Graphical User Interface
I/O	Input/Output
IDA	Institute for Defense Analyses
OO	Object-Oriented
OOT	Object-Oriented Technology
SAMeDL	SQL Ada Module Description Language
TGMS	Tiered Geometric Modeling System\s